SLANC: A SPECIFICATION LANGUAGE FOR COMPUTABLE CONTRACTS

BY

TURKI SAUD ALHAZMI

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

MAY 2018

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **TURKI SAUD ALHAZMI** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

Thesis Committee

Dr. Musab A. Alturki (Adviser)

Dr. Moataz Ahmed (Member)

Dr. Mohammad Alshayeb (Member)

For:

Dr. Khalid Aljasser
Department Chairman

Dr. Salam A. Zummo
Dean of Graduate Studies

2/8/2018
Date

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

Dedication

To my great **father** and my wonderful **mother**.

ACKNOWLEDGMENTS

I would like to thank my adviser Dr Musab Alturki for his guidance and understanding throughout this thesis. Also, I would like to thank the committee members Dr Moataz Ahmed and Dr Mohammad Alshayeb for their insightful comments on this thesis. Finally, I would like to thank my family for their support and patience throughout my studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

NAME: Turki Saud Alhazmi

TITLE OF STUDY: SlanC: A Specification Language for Computable Contracts

MAJOR FIELD: Computer Science

DATE OF DEGREE: May 2018

Recently, there has been a strong resurgence of interest in automated legal reasoning due primarily to rapidly increasing legal costs and recent advances in computational logics and language-based analysis techniques. As contracts are central to computational law, automating verification and synthesis of contracts has the potential to significantly reduce costs, increase productivity and minimize chances of legal disputes. However, any attempt to achieve this automation fundamentally requires a (formal) computable model of a contract. Despite their ubiquity and significance, contracts are, in practice, still typically written in natural languages, such as English. However, natural languages cannot be interpreted by machines reliably, which makes it difficult to introduce any kind of automation to some of the contract lifecycle management activities, which is why many models were

proposed in the literature to formalize contracts, allowing automated analysis of contracts to some extent. However, specifying contracts in these formalisms is difficult because they expressed them in languages that are too complex for non-expert users such as lawyers. Furthermore, specifying contracts in most of these formalisms offered little to no return on investment, because they had no way of automating any kind of reasoning, as they were not executable. Motivated by this, we propose and develop a new formal language for specifying contracts, SlanC*, a* ***S****pecification* ***lan****guage for computable* ***C****ontracts. It is uniquely characterized by being close in appearance to a natural language, increasing its potential accessibility to non-experts, while still being structured and formally well-defined, enabling automated formal reasoning about contracts. We developed a prototype of this language with its violation detection semantics in the Maude formal environment and simulated several small examples and real world case studies. Furthermore, we evaluated the expressiveness of* SlanC *against a set of requirements for contract specification languages which showed that* SlanC *satisfies the basic requirements for specifying contracts. Also, we evaluated the performance of its Maude specification showed that there is a linear correlation between the number of processed events and the number of operations performed by Maude for a given contract. The results from the simulations and the evaluation show that developing domain specific languages for writing contracts could potentially enable automation of all operations on contracts.*

ملخص الرسالة

الاسم: تركي سعود سليمان الحازمي

عنوان الدراسة: سلانك: لغة لتوصيف العقود المحوسبة

التخصص: علوم حاسب آلي

تاريخ الدرجة العلمية: مايو 2018

في الأونة الاخيرة ازداد الاهتمام بشكل كبير في أتمتة المنطق القانوني بسبب الارتفاع السريع في التكاليف القانونية من جهة، والتقدم الملحوظ لتقنيات حوسبة المنطق وتقنيات احليل اللغات الطبيعية من جهة اخرى. أهمية العقود في القانون جعلت منها محورا رئيسيا لكثير من البحوث في مجال حوسبة القانون، حيث ان جل هذه البحوث يتمركز حول أتمتة عمليتي توصيف وتحليل العقود والتي من الممكن ان تؤدي الى خقض تكاليفها والتقليل من اختمالية تنازع المتعاقدين. إن أتمتة عمليتي توصيف وتحليل العقود تتطلب وجود نموذج رسمي (مبني على أسس رياضية) للعقود والذي سيمكن من توصيفه وتحليلها رياضيا، ولكن وعلى الرغم من اهمية العقود وشيوعها الا انها مازالت تكتب باللغات الطبيعية مثل العربية والانجليزية. وهذه اللغات - رغم سهولتها للإنسان - الا ان الحاسوب لايستطيع فهمها وتحليلها بكفاءة عالية يمكن الاعتماد عليها في فهم العقود وتحليلها آليا. لذلك قدمت العديد من البحوث نماذج رسمية مختلفة لتوصيف العقود بهدف حوسبة العمليات المختلفة على العقود، ولكن بعضا منها استخدمت لغات صعبة في كتابة العقود والبعض الآخر طور نموذجا لايمكن حوسبته(أتمتة أي من العمليات المتعلقة بالعقود حين يستخدم لتوصيف العقد). بدأً من هذا المنطلق وفي هذه الرسالة نقوم بتطوير لغة رسمية لكتابة العقود نطلق عليها اسم (سلانك) وهي اختصار لجملة إنجليزية تعني لغة توصيف العقود المحوسبة. إحدى اهم خصائص هذه اللغة هي رسميتها ومشابهتها للغة الانجليزية وهذا "التشابه" نتوقع أن يساعد في تسهيلها للمتعلمين بينما تمكن رسمية اللغة من أتمتة العمليات على العقود في الوقت ذاته. قمنا بتطوير نموذج لهذه اللغة ومعانيها باستخدام بيئة "ماود" والذي مكننا من محاكاة مجموعة من العقود البسيطة ومجموعة اخرى مستوحاة من عقود دارجه. بعد

ذلك قمنا بتقييم هذه اللغة بالاعتماد على مجموعة من المتطلبات الأساسية للغات كتابة العقود والذي آستنتجنا من أن اللغة تستوفي المتطلبات الاساسية لكتابة العقود. بالإضافة الى ذلك، بين لنا تقييم هذه اللغة ونموذجها ان تطوير لغات رسمية خاصة لتوصيف العقود سيمكن من أتمتة جميع العمليات على العقود.

CHAPTER 1

INTRODUCTION

Computational law is the science of making law computable so that legal artifacts, including contracts, regulations and policies, can be made amenable to mechanized analysis and synthesis. This field is not new; in fact, early attempts dating back to the late 1950s have been reported [4]. However, recent years have seen a strong resurgence in interest in automated legal reasoning, which can be attributed primarily to: (1) the rapidly increasing legal costs and conflicts in recent years, (2) increasing ubiquity of intelligent systems (especially pervasive computing systems like IoT), and (3) the recent advances in computational logics and language-based analysis techniques.

A central component of computational law is a contract. Contracts are integral to the execution of processes. They are also heavily used as tools to communicate regulations, agreements and memorandums of understanding. Generally, a contract records a legal agreement between two or more parties. It contains a set of statements, which may include rules, conditions, permissions, prohibitions, obligations, governing laws, actions, deadlines, time stamps, etc. According to

Brian Blum [5], a contract must (1) contain an agreement between at least two parties, (2) contain at least one promise made by the parties, (3) establish an exchange relationship between the parties, and (4) be legally enforceable.

Contracts are important because of the following reasons:

- They are legally enforceable in a court of law.
- They provide individuals and businesses with a legal document stating the expectations of both parties and how negative situations will be resolved.
- Contracts often represent a tool that companies use to safeguard their resources.

1.1 Motivation

Despite their ubiquity and significance, contracts are, in practice, still typically written in natural languages, such as English. However, natural languages cannot be read and interpreted by machines reliably, which makes it difficult to introduce any kind of automation to some of the contract lifecycle management activities. These include the following:

- Drafting: Difficulty to implement tools to provide support to contract writers during the contract drafting process, because they do not follow a standard format. These tools could be as simple as autocomplete or intellisense[1].

[1] similar to what programmers have with popular programming languages, where suggestions are provided as program statements are written.

Another issue, is checking for basic conflicts as the contract is being typed, which is highly complex with a natural language.

- Execution: Having an interpreter that can interpret contracts and enable experimentation and simulation on it.

- Monitoring: Automate the process of detecting run time violations and checking conformance with contract requirements

Automating these activities would also provide a foundation that can be built on to automate reasoning about other properties of contracts such as consistency analysis and checking conformance of contracts with general regulations and policies. Therefore, it's crucial to have a formal language for specifying contracts, which makes contracts machine-readable and understandable .

That is why many formalisms were proposed for contracts that aim to alleviate some of these problems, especially ones related to checking conformance and run-time monitoring. These solutions were very successful in capturing those aspects, however, they were difficult to use because they expressed contracts in languages that are too complex for non-experts. These non-experts are ironically, the people[2] who are tasked with writing and understanding natural language contracts. Moreover, even if they could specify contracts in these languages, there was little to no return on investment, because some of these were unexecutable. Figure 1.1 highlights the complexity involved in specifying a contract in one of these languages[3], by showing the original contract and it's specification, which

[2]lawyers, judges etc

[3]The syntax of the language is inspired by deontic and propositional logics

The American call option gives the buyer the option to buy one share of stock at some time in the future from the seller [1]. It is defined as follows:

1. If the buyer buys the option, which expires on April 1st, 2018, to purchase one share of stock by transferring $5 on October 1st 2017 then he can exercise that option at any time in that period, inclusive. The strike price of the option is $80.
2. If the buyer chooses to exercise that option by paying the strike price on January 1st 2018, then the seller must transfer one share of stock to the buyer within 30 days of the payment date.

$C = (t_{offer}, D_1, D_2, R_1)$
$D_1 : t_{buy} = 0(June30, 2015)$
$D_2 : t_{expire} = 170(December17, 2015)$
$R1 = \phi_1? \rightarrow R_2$
$\phi_1 = obs - event(X_{buy}, e_1)\ X_{buy} = t_{buy}$
$e_1 = (\text{“}BuyOption\text{”}, transfer(\$5), buyer, seller)$
$R_2 = \phi_2? \rightarrow D_3, A$
$_2 = obs - event(X_{time}, e_2)\ t_buy\ X_{time}\ t_{expire}$
$e_2 = (\text{“}ExerciseOption\text{”}, transfer(\$80), buyer, seller)$
$D_3 : t_{exercise} = X_{time}$
$A = O(e_3, [t_{exercise}, t_{exercise} + 30])$
$e_3 = (\text{“}TransferStock\text{”}, transfer(stock), seller, buyer).$

Figure 1.1: American call option original contract (Top) and its specification (bottom) as seen in [1]

were presented in [1]. The fundamental problem with these languages is that specification of the contract in the language is a concise logical formula that looks nothing like the original contract, which inhibits readability and writability. They are also not directly executable, relying mostly on manual analysis methods.

Motivated by this, we propose a new formal language for writing contracts. We design the constructs of the language to resemble natural language, which we anticipate will minimize its learning curve for general audiences. Furthermore, once contracts are specified in the language, they can be automatically reasoned about. This lays the foundation for a future generation of autonomous, self-

adapting contracts. We call this new language SlanC, a **S**pecification **lan**guage for computable **C**ontracts.

1.2 Problem Description

Given the unsuitability of natural languages and limitations of existing logics for contract specification, is it possible to design and develop a language that has the following characteristics:

- enables natural specification of contracts like natural languages.
- is formally definable like logic based formalisms.
- is executable enabling automatic formal analsysis.

More specifically, we are trying to answer these two research questions:

- [**RQ1**]: How is it possible to define and implement a formal language for specifying contracts from different domains?
- [**RQ2**]: How can the executable version of that language be used to reason about contract violation, i.e automatically detect violations of contracts through formally defined semantics?

1.3 Methodology

We follow an agile approach in the development of this language, which aims to incrementally enhance SlanC by maximizing its expressiveness and minimizing its

learning curve. Naturally, these are conflicting requirements, so we have to strike a balance between them. We do that, by ensuring that any new construct we introduce to SlanC is intuitive and that it adds to the expressiveness of it.

Since we follow an agile approach, we start with a basic set of requirements and as we develop them new requirements might arise that we are not aware of, which we will address accordingly.

After developing these basic requirements, we move on to what we call "iterative language enhancement by example". Here we start to test SlanC by trying to specify real world examples of contracts, if we succeed then we simulate those examples. If we fail, then we try to identify whats missing form SlanC that if we add, will allow us to express and simulate that example. As we do this with more and more real world examples, we achieve three important things:

- Iteratively enhance the repertoire of SlanC, to make it more expressive.
- Increase confidence in the expressiveness of SlanC, because we are essentially building a test suite of real world examples for it.
- Increase confidence in the correctness of the formal semantics of SlanC, because contracts are simulated under different circumstances.

1.4 Contributions

In this section we highlight the main contribution of our work, which include:

- A catalog of abstractions that are essential to naturally capturing contracts,

synthesized through investigation of several examples.

- The design and implementation of a formal language SlanC, a **S**pecification **lan**guage for defining computable **C**ontracts, that allows close to natural specification contracts.

- Formal algebraic semantics for run-time violation detection.

- Excutability and analysis tool that enables automated detection of contract run-time violations.

1.5 Background

In this thesis, we use Maude to implement SlanC and its semantics. This section presents an overview of Maude and rewriting logic and a study about why it was chosen as the formal environment to implement SlanC.

Maude [6] is a high-performance rewriting engine that supports both equational [7] and rewriting logic [8] specifications. These logics are computational and allow for naturally specifying a very wide range of systems ranging from simple sequential languages to very complex concurrent and real-time systems.

An equational theory, which is a tuple of the form $(\Sigma, E \cup A)$, gives a description of a (deterministic) system constructed by the symbols declared in the signature Σ and whose properties are determined by the set of equations E and equational attributes A (such as commutativity and associativity). An equational theory is modeled in Maude by a functional module, which may include sort and subsort

declarations, operator declarations, and possibly conditional equations. Sorts are introduced in a module using the `sort` and `sorts` keywords. The subsorting relation can be defined using the `subsort` keyword. Operators in Maude can be declared using `op` in mixfix notation, where the locations of the operands of an operator are user-definable. For example, the declaration: `op _+_ : Nat Nat -> Nat`, defines a binary operator `+` with the two operands appearing to the left and right of the symbol `+`. The declaraion also defines the sorts of the two operands and that of the result (all of the sort `Nat` for naturals). Unconditional (resp. conditional) equations in Maude are introduced using the keyword `eq` (resp. `ceq`). A conditional equation has the form `t = t' if C`, where `t` and `t'` are terms and `C` is an equational condition. Functional modules in Maude are executed using equational simplification, by orienting the equations from left to right. More details about Maude and equational simplification can be found in [6] .

Many papers in the literature used Maude to build executable models for different systems. In [9], the authors specified a network flow algorithm, which is called Flow Based Adaptive Routing (FBAR) introduced in [10] using the Packet Language for Active Networks (PLAN) [11]. The authors used their specification to prove the correctness of the algorithm in a given scenario.

The authors in [12] explored the use of Maude in Domain Specific Modelling (DSM). DSM is a way of modelling systems through the development of Domain Specific Languages to model the different parts of the system. The authors stipulated that current approaches focus on developing the structural aspects without much focus on defining the semantics. The authors argue developing behavioural

semantics is crucial because it makes models amenable to formal analysis. The authors proposed using Maude as a formal environment for specifying Domain Specific Languages. They argued that Maude's excutability allows for applying a wide range of formal analysis techniques with ease.

The authors in [13] gave precise semantics for ATL using Maude. ATL (ATL Transformation Language) is a model transformation language in the field of Model-Driven Engineering (MDE), which provides ways to produce a set of target models from a set of source models. They address two critical problems that exist in the model transformation space. First, their formal semantics give an account of the expected behavior of ATL. Second, the executable Maude specification allows for automated verification and debugging for model transformations that are too complex to verify manually for correctness.

The aforementioned works, illustrate the versatility of Maude as a semantic framework and its ability to specify anything from simple algorithms to languages. Furthermore, a cornerstone feature of Maude that makes appealing for use as a semantic framework is its executability, which allows for automating formal analysis.

CHAPTER 2

LITERATURE REVIEW

In this chapter we present the results of the literature review we conducted about the subject of this thesis. The works that we reviewed can be classified into 2 main categories, surveys and formalisms. Sections 2.1 and 2.2 present the surveys and formalisms, respectively.

2.1 Surveys

Elgammal et al [14] produced a comparative analysis of 3 formalisms that can be used to specify regulatory compliance, namely, LTL, CTL and FCL. LTL stands for Linear Temporal Logic which belongs to the temporal logic family of languages, where each state has only one possible future. CTL or Computational Tree Logic which belongs to the same family of languages but its model of time is different, where each moment in time has many possible futures. FCL or Formal Contract Language which is based on defeasible logic and deontic logic of violations. The comparative analysis for these involved specifying a loan approval scenario ,which

was presented in [9], to each language and then comparing those specifications against a set of 11 features that they deemed necessary for any Compliance Request Language. Namely, Formality, Usability, Expressiveness, Declarativiness, Consistency Checks[1], Non Monotonicity[2], Generic[3], Symmetricity[4], Normalization[5], Intelligible feedback [6] and Realtime Support . CTL,LTL and FCL were formal, declarative and have realtime support, while being partially expressive. Furthermore, FCL was found to be the only language that supports normalization and consistency checks, while having partial support for symmetricity and no support for intelligible feedback. For non-monotonicity CTL had partial support and FCL had full support while LTL had no support. For the rest of the features all of them were found to be on equal grounds. Finally, the authors concluded that choice of best formalism is context dependent and that temporal logic based formalisms have more advantages over others for specifying regulatory compliance.

Hivted [15] presented a survey of existing formalisms and models for contracts. Furthermore, the author identified a set of requirements and argued that any model should support, to be able to specify contracts effectively. These requirements are: "(R1) modelling of contract participants; (R2) parametrized contract templates; (R3) (conditional) commitments, i.e., obligations, permissions, and prohibitions; (R4) absolute and relative temporal constraints; (R5) history-

[1]language provides facilities for detecting and resolving inconsistencies

[2]a violation of a rule doesn't necessarily mean an error(i.e rules are not rigid)

[3]expresses the various types of compliance rules

[4]annotate business process modules with compliance rules

[5]remove redundancies

[6]not just identifying violations but also why they happened and how to resolve them.

sensitive commitments; and (R6) basic arithmetic; (R7) contrary-to-duty (reparation clauses); (R8) potentially infinite and repetitive contracts; (R9) compositionality; (R10) deterministic contract execution (run-time monitoring); (R11) blame assignment; (R12) the isomorphism principle; and (R13) subject to analysis". The author also argued that these might not be sufficient to capture all types contracts, however, they represent core aspects of them. Moreover, authors categorized current models(at the time) into three categories: (deontic) logic based formalisms, event-condition-action based formalisms and action/trace based formalisms.

2.2 Contract Formalisms

This section discuss the contract formalisms that we encountered in our review, which can be roughly categorized into two broad categories, namely, formalisms that are based on logic and formalisms based on programming languages. Sections 2.2.1 and 2.2.2 present them, respectively.

2.2.1 Formalisms Based on Logic

The formalisms in this category are characterized by having been built on and specified using some form of logic (mostly deontic). The reason why deontic logic forms the basis of many of these formalisms is because it is the logic that deals with normative constructs giving precise definitions to concepts like obligations and prohibitions. Furthermore, since deontic logic lacks the ability to describe some aspects, many formalisms tend to extend other logics in addition to it. For

exampe, deontic logic cannot express time constraints, hence, many formalisms extend temporal logic enable their specification.

Governatori et al [16] presented a formal system (FCL), based on the deontic concepts of obligations, prohibitions and permissions, for specifying contracts. This formal system has formal syntax and precise semantics for analyzing and reasoning about contracts written in it. This system is used as a foundation for Business Contract Language (BCL), for writing contracts and enabling event-based execution monitoring of them. Formal mappings are, also, given from BCL to FCL and vice versa. Overall, BCL has high expressiveness and it has features that facilitate contract monitoring in an organizational context.

Pace et al [17] presented a case study where they translate a conventional contract between an internet service provider and a client into a contract specification language CL. Then they translate that specification into an extended μ-calculus (cμ) where they obtain the semantics as transition system with labels. Finally, they model check the contract by translating its cu specification into input to NuSMV model checking tool. Using that, they prove that there are some ambiguities in the aforementioned contract and correct those ambiguities based on output from NuSMV.

Prisacariu et al [18] presented a formal language for writing contracts, that combines logical and automata like approaches. The syntax of the language influenced by that of deontic, temporal and dynamic logics, and it has formal semantics given through the encoding to μ-calculus. The language captures core aspects of contracts, such as, obligations, prohibitions and permissions. Furthermore, a con-

tract in the language is composed of two main sections, a set of definitions $\mathcal{D}$ and a list of clauses $\mathcal{C}$ which contains obligations, prohibitions and permissions defined over actions given in $\mathcal{D}$. Key features of CL is that: (1) it's Decidable. (2) it avoids the most important logical paradoxes coming from deontic logic, namely: (1)Ross's paradox. (2)The Free Choice Permission paradox. (3)Sartre's dilemma. (4)The Good Samaritan paradox. (5)Chisholm's paradox. (6)The Gentle Murderer paradox.

Pace et al [19] proposed using controlled natural language as an interface to make specifying contracts in deontic logic more accessible to a general audience. Furthermore, they presented a deontic logic based formalism for specifying contracts, along with some remarks on a possible controlled natural language to be used. Finally, they presented some examples of how natural language contracts can be reduced to controlled natural language that are parsable and syntactically simpler and semantically clearer.

The authors in [20] presented a new version of the contract language CL. The version they presented is based on deontic logic and propositional dynamic logic and it has precise direct semantics in terms of normative structures. It captures the basic aspects of any contract, such as obligations and prohibitions and it features operators that capture more complex scenarios such as sequencing, concurrency and choice. Furthermore, the language applies logic modalities exclusively over actions which eliminates many of the logical paradoxes present in Standard Deontic Logic. Overall, the language they present is rich with constructs that could be used to specify general contracts, however, its not clear how it fairs in

specifying contracts from special domains, e.g financial.

Hivted et al [21] proposed an abstract trace-based model for multiparty contracts that supports blame assignment, where a contract is viewed as a function that maps an action traces to a verdicts. The action trace is a set of all events that occurred in the environment of the contract ordered by time and the verdict is a tuple of the current time and the set of all parties to blame when there is a breach. The model supports composition of contracts through conjunction and disjunction. Furthermore, they propose CSL, a declarative contract specification language with formal semantics that simplifies expressing contracts using their abstract model. It supports the essential components for expressing any contract, such as relative and absolute temporal conditions, obligations and permissions. However, it lacks a way of describing a prohibition directly, instead, they express it as an unfulfillable obligation which is unnatural to say the least. Moreover, they develop an algorithm that monitors contracts specified in CSL at runtime. Finally, the language was used in the paper to successfully specify real world examples of contracts, such as a lease agreement and a sale of goods contract.

Angelov et al [22] presented a framework (AnaCon) for analyzing conflicts in normative texts written in controlled natural language(CNL) that they also proposed. AnaCon translates normative texts written in CNL to CL syntax and the uses the CLAN tool to model check those texts for conflicts and reports back with them. These conflicts are presented to the users in CNL, which makes corrections to these texts simpler, as the users do not have to understand the complexities of CL. The proposed CNL is simple enough to be used by a nontechnical person and

expressive enough to capture core aspects of normative texts. However, it doesn't cover all the components of CL, which might inhibit expression of some types of normative texts.

Cambronero et al [23] presented a calculus for reasoning about contracts that adopts an action based view, where contracts are viewed as constraints regulating the behavior of a set of agents running in parallel. Furthermore, the authors introduce the notion of agent and system evolution where an agent or system "evolves" after performing a certain action, subject to constraints about the agents and actions, e.g the system A evolves to A' iff agents (a,b) synchronize on action c. Their calculus uses the deontic modalities of obligation, permission and prohibition as basis for the calculus and it supports conditionals, reparation clauses and recursive contracts (repetition). Also, the authors gave operational semantics for the calculus and showed how contracts written in in the calculus can be verified at run time using LARVA [7].

The authors in [1] proposed a formal language for specifying general contracts that has mathematically precise semantics. The language is based on simple type theory and it models contracts as a set of time sensitive conditional and unconditional agreements, which depend on observables from the environment of

[7] is an automaton based tool which enables the runtime verification of Java programs against specifications written using DATEs (Dynamic Automata with Timers and Events) — finite state automata extended by various features, including symbolic state, automata communication, timers and dynamic replication. For the scope of this paper, we limit the features of the specification automata to the ones we require for contract monitoring. The authors translated showed how to automatically translate the specifications in their calculus to LARVA syntax.

the contract. These agreements change the state of the contract when satisfied or breached, which is how the meaning of the contract is captured. The notion of contract state makes the process of capturing the meaning of dynamic and static contracts more intuitive (i.e through changes in state). Furthermore, many of the aspects of contracts are expressed intuitively by the language, such as obligations and prohibitions. However, the mathematical nature of its syntax makes any contract expressed in it, significantly different from the actual, in terms of representation.

2.2.2 Formalisms Based on Programming Languages

The formalisms that belong to this category have one feature in common, in that they are embedded within a programming language. Therefore, they inherit all the capabilities and drawbacks of that language. Most of the formalisms that we have seen in this category are implemented using Haskel.

Jones et al [2] proposed using concepts from functional programming to specify financial contracts. The authors also introduce a combinator library built using Haskell that could successfully specify financial contracts in a compositional manner. The idea is to have a small set of primitive combinators that could be used to express a variety of contracts. The interesting thing about this library in particular is that contracts are specified in compositional manner, implying that a basic building block of a contract (e.g obligation) is considered a contract. These basic building blocks are then combined using well defined combinators to produce a new contract. Furthermore, the authors also give compositional denotational

semantics for contract valuation, for contracts specified using their library.

The author in [24] proposed a formal language (POL) for privacy options that is built on the language for specifying financial contracts proposed in [25]. At its core the language expresses contracts that specify the rights as given by the owner of some data to a user and the obligations that are attached to them. Its syntax is similar to Haskell and it describes privacy options concisely. Also, the paper discusses the semantics of managing specifications written in POL and ways of extending it.

The authors in [26], presented a language for specifying financial derivatives that is independent of pricing models. The language is based on the one described in [27] so its syntax and semantics are fairly similar to it. Furthermore, the papers main contribution is presenting a rich set of constructs for specifying observables, which are crucial to capturing the dynamic aspect of financial derivatives. For example, specification of two observations that must happen at the same time.

The authors in [28] presented a symbolic framework for financial contract management that is capable of expressing financial derivatives. Its Domain specific language allows writing multi-party contracts in a compositional manner with syntax similar to Haskell and precise denotational cash-flow semantics. The semantics are given with respect to the contract environment, which is a list of observations about the parties and the clauses in the contract. The semantics transform those observations to a set of cashflows between the parties of the contract over time. Furthermore, The authors gave reduction semantics for the language, which evolve the contract over time. Moreover, the language was formalized in the Coq the-

orem prover which was used, among other things, to generate certified Haskell implementation of their contract management framework.

2.3 Formalisms based on XML

All of the works that we encountered in this category are improvements on Legal-RuleML [3]. In that paper, the authors describe an extension to RuleML that defines all the domain specific construct pertaining to contracts, which allows specification of contracts declaratively using XML.

Compared with SlanC, all aforementioned languages and encodings are, by their nature, low-level, requiring the contract writer to be knowledgeable in formal languages and experienced in formally specifying systems. This divorces the representation of the contract in the formalism from that of the actual contract, making the language inaccessible to non-experts. Furthermore, many of these of formalisms lack executable semantics, mainly because their underlying formalisms are not computational. In other cases, executability is achieved through implementations (like in Haskell [29]). On the other hand, SlanC simplifies specifying contracts by minimizing the representational gap between the actual contract and the one written in the language. It achieves this by abstracting away how the contract should be evaluated from the contract writer by leveraging intuitive and semantically rich constructs with formal semantics that precisely describe how the evaluation is done. Furthermore, SlanC is readily executable through its formal algebraic semantics in Maude.

Another approach for formal specification of semantics of Domain Specific Languages is defining a semantic mapping between the new language and an existing one that already has formal semantics. This semantic mapping is a translation from the expressions of the new language to the language that has formal semantics. The reason we did not consider this approach for SlanC is two fold; First, lack of languages for specification of contracts that have formal semantics defined. Second, languages that have formal semantics are either unexecutable or they are not as easily accessible as Maude for developing domain specific languages.

CHAPTER 3

SLANC: A SPECIFICATION LANGUAGE FOR COMPUTABLE CONTRACTS

This chapter introduces SlanC, a structured **S**pecification **lan**guage for defining computable **C**ontracts. The language incorporates high-level constructs and structures that represent the different abstractions needed in the specification of contracts given in Section 3.1.

Fundamentally, SlanC is a sequential, domain-specific programming language for contracts. Specifications of contracts in SlanC are evaluated in sequence from top to bottom. SlanC is also deterministic, so that the same contract always yields the same state under the same circumstances. Although it allows for an apparently declarative style of specification, SlanC is foundationally an imperative language in disguise; the operational meaning of a contract in SlanC is defined by the changes

it makes to its state. Moreover, traditional variable declarations and assignment statements that are typically found in an imperative language are not explicitly given by SlanC. Instead, they are more naturally embedded in the specification of rules where they may be needed in a contract, adding to the declarative feel of the specification without sacrificing expressive power.

SlanC is uniquely characterized by having syntax that is well structured, yet closely resembles the variations of natural language that are used to describe contracts. This makes the language potentially more accessible and usable by a broad range of users, experts and non-experts alike. It also helps streamline the process of translating natural language contracts to and from SlanC. Furthermore, the constructs of SlanC are semantically rich, allowing the writer to express fairly complex rules in a compact and precise manner. Moreover, SlanC is generic in that it allows for expressing user definable events and actions.

This chapter is organized as follows: Section 3.1 presents a set of abstractions that form the basis of SlanC and Section 3.2 present an informal description of the syntax and semantics of SlanC.

3.1 Abstractions

There are some basic contract building blocks that are common to all contracts. This section introduces these building blocks in the form of abstractions and idioms. Having a clear definition of these abstractions is necessary to defining a formal language for contract specification while maintaining relevance. The

The contract is between a client and an Internet service provider. The provided service depends on the level of Internet traffic used by the client. There are two levels of traffic: *high* and *low*. This example considers only the following clauses of the contract:

1. While the Internet traffic is low, the client must pay $X. Traffic is higher than XX Mbps then the client must pay XX$.
2. Whenever the Internet traffic is high, the client must pay $Y and lower traffic to low immediately.
3. In case the client wants to delay the payment, it is essential that the client notifies the service provider by e-mail specifying that the client will pay later.
4. Whenever the client delays the payment without notification, the client must immediately lower the Internet traffic to low, and pay later $2Y.
5. If the client does not decrease the Internet traffic to low immediately, the client will have to pay $3Y.

Figure 3.1: Internet Access Services Contract

abstractions are illustrated below using a typical example that we borrow from [18] shown in Figure 3.1.

3.1.1 Contract

A contract is a self-contained legal artifact. It encapsulates the actual conditions and agreements in addition to all the basic information needed for the contract to be meaningful. The contract records a legally binding agreement between its parties. In the example above, the contract states an agreement between a client and an Internet provider. The contract begins by introducing terms (e.g. the parties, low and high traffic levels, etc) and then uses these terms to define exactly the agreement details. Within the contract, promises and obligations, such as the client promising to pay for the service, and conditions governing them are defined. Therefore, a contract must encapsulate both the preamble information

such as parties, dates and definitions along with the actual agreement statements, while clearly distinguishing them from each other.

3.1.2 Preamble

A preamble is a set of statements at the beginning of a contract that contains basic definitions and necessary information about the contract, such as start and end dates and the parties of the contract. In the above example, the main pieces of information given in the preamble are:

- The title of the contract, which is Internet Access Services,
- The parties, who are the client and the Internet service provider,
- A description of the contract stating what the contract is for, and
- Terms and their definitions, e.g. high and low levels of traffic.

In general, the preamble contains much more than what is shown in this example, including start and end dates and more precise definition of services (e.g. defining exactly what low and high means in terms of amounts measured in, say, Mbps). The preamble typically also sets the context of the contract by making appropriate references to other relevant contracts or regulations.

3.1.3 Agreements

An agreement is a statement that compels one or more of the contract parties to do (or to refrain from doing) something for one or more other parties. The agreement

may express an obligation or a prohibition. The Internet Access Services example states some agreements, such as:

- "the client must pay $Y and lower traffic to low immediately".
- "it is essential that the client notifies the service provider by e-mail specifying that the client will pay later".
- "the client must immediately lower the Internet traffic to low, and pay later $2Y".

As can be seen from these examples, an agreement typically specifies actions and amounts, refers to terms previously defined (in the preamble), and involves temporal references and time requirements.

3.1.4 Rules

Agreements are typically subjected to conditions, specifying when the agreement's obligation, prohibition or permission must be satisfied. Rules enable the specification of conditional agreements. A rule is composed of two main components: a condition and a consequent. The condition defines an assertion that is either true or false, and the consequent is only considered when the condition is true. We highlight below some sample rules from the example above, showing their condition and consequent parts:

- *Condition*: "the Internet traffic is high"

  Consequent: "the client must pay $Y and lower traffic to low immediately"

- *Condition*: "the client delays the payment without notification"

  Consequent: "the client must immediately lower the Internet traffic to low, and pay later $2Y".

- *Condition*: "the client does not decrease the Internet traffic to low immediately"

  Consequent: "the client will have to pay $3Y".

It is important to note that the consequent of a rule can in general contain a list of other rules and agreements resulting in complex rules.

3.1.5 Environment

The environment of a contract is crucial to the discussion of any language for contract specification, because almost all the clauses (rules) in a contract depend on observable actions obtained form its environment. For instance, the Internet Access Service contract contains assertions about users bandwidth usage (e.g. a client exceeding the traffic limit), which depend on observations from the contract's environment. Furthermore, these observables are usually timed, such as the agreement that a client makes a payment immediately. Although the environment of a contract is separate from the actual contract, capturing the environment abstraction in the language design is fundamental for enabling analysis and verification of dynamic behaviors of contracts in the future.

3.1.6 State

The state of a contract stores information about the agreements in that contract. Ideally, the state should contain information about:

1. Status of each agreement (fulfilled/broken) and time of the status change relative to the environment of the contract.

2. List of all transactions that are time stamped.

3. List of all assets and their current status.

.

3.1.7 Time

Time is a crucial component of any contract, because most (if not all) contracts are time sensitive. That is, they are bounded by start and end dates and their clauses have temporal conditions that specify when they hold. Therefore, a simple and readable representation of time is required to facilitate using it in the context of a contract.

3.1.8 Asset

An asset is any thing that can be owned and transferred from one owner to another. Contracts usually deal with assets directly by specifying a transfer operation that must occur at some time, e.g the payments in the internet access services contracts

where the asset is money owned by the client and transferred to the provider. We abstract an asset as a 4-tuple that has a name, a code, supply and an owner.

3.1.9 Containers

A container is an abstraction on composition operations, e.g AND, OR , XOR & NOR. Usually composition operators take two parameters, unlike unlike containers which take arbitrary number of parameters. The advantages of having containers are: (1) simplicity and ease of use. (2) ability to extend these containers easily to incorporate more semantic information. An example of how these containers are more extensible is illustrated through the example in section 3.2.10 .

3.2 Syntax and Informal Semantics

This section presents the syntax and informal semantics of SlanC. We start by describing contracts and work our way through to to smaller components that make them up.

3.2.1 Contract

The abstraction of a contract maps directly into SlanC as a contract module, which that encapsulates all other components of a contract. The contract module is composed of two smaller components, namely, the preamble and body. Therefore, a contract in SlanC has the following form:

```
Contract {
```

```
    [PREAMBLE]

    [BODY]

}
```

The preamble and body resemble the ones that exist in actual contracts. We discuss each of them in detail in upcoming sections.

3.2.2 Preamble

SlanC also defines a preamble, which is the component that contains basic information about the contract. Initially, SlanC supports the following most fundamental properties: 1) Contract Name, 2) Contract Description, 3) Parties, which is a comma separated list of names of parties, 4) Start Time, 5) End Time and 6) Extends field which contains the list of names of all contracts that this contract inherits properties from. We will discuss the parties and the extends field in more detail later, but for know focus on the Internet Access Services example presented in Section 3.1, a preamble for such a contract might look like the following:

```
Preamble {
    Name: "Internet Access Services"
    Parties: [LIST OF PARTIES]
    Start Time: (1 , 1 , 2018)
    End Time: (1 , 4 , 2018)
    Extends: [LIST OF PARENT CONTRACTS NAMES]
}
```

3.2.3 Contract Parties

A party is foundationally a 4-tuple that has the following fields:

- Party ID: an identifier for the party that can be used to identify this party in other constructs in SlanC . The id of the party has single quote at the beginning.
- Party Name: a string representing the name.
- Party Description: a string describing the party.
- Party Address: a string representing the address of the party.

In SlanC a new party is defined as follows:

```
{id: 'Landlord , name: "John Smith"
    ,description: "Owns a building"
    ,address: "123 Street, Small town"}
```

A list of parties can also be constructed by separating different parties with white spaces, as follows:

```
{id: 'Landlord , name: "John Smith"
    ,description: "Owns a building"
    ,address: "123 Street, Small town"}


{id: 'Tenant , name: "Jane Doe"
    ,description: "Want to rent a building"
    ,address: "456 Street, Small town"}
```

3.2.4 Assets

The abstraction of an asset maps to an asset component in SlanC. It is a 4-tuple that has the following information:

- Asset name: a string that contains the asset name.
- Asset code: Used to identify the assets in the specification of transfer operations, which will be discussed later.
- Asset Supply: The total available amount of this asset.
- Owner ID: The owner of an asset is a

In SlanC a new asset is defined as follows:

```
{name:"Saudi Riyal" , code: SAR
    , supply: 1000000 , owner: 'Landlord }
```

This states that the owner has 1000000 Saudi Riyals. Furthermore, a single party can have more than one asset and many parties can have the same type of asset identified by its code. A list of assets can be constructed by separating different assets with a white space, as follows:

```
{name:"Saudi Riyal" , code: SAR
    , supply: 1000000 , owner: 'Landlord }
{name:"Saudi Riyal" , code: SAR
    , supply: 1000 , owner: 'Tenant }
```

3.2.5 Time and time references

In SlanC time is represented by a 3-tuple of positive integers (DD,MM,YYYY), that represent the day, month and year respectively. Any two times can be added or subtracted using the operators (++) or (–) respectively. The following are examples of times and operations on times:

```
(1 , 1 , 2018)

(2 , 2 , 2018) ++ (0 , 0 , 4)
```

In the second example we add four years to (2 , 2 , 2018) to obtain (2 , 2 , 2022).

A time reference is a string that refers to a well known time, i.e if that string is evaluated in the state of the contracts it will result in an actual time. Time references can also be added to or subtracted from using the operators (+) and (-), as follows:

```
"payment time" + (1 , 3 , 4)
```

This states that the time referred to by payment time should be extended by 4 years 3 months and 1 day.

3.2.6 Actions

Actions specify an act that is performed by one party for another. In SlanC there are two types of actions:

- An arbitrary string action is a string specifying an arbitrary event, such as ”say thank you” and ”remove obstacle”. SlanC doesn't understand this

action, so to check for any assertion about it we use equality. This entails that SlanC will not be able to track the meaning behind that arbitrary string. The following is an example of this type of action:

```
"say thank you" by 'Landlord for 'Tenant
"remove bandwidth cap" by 'ISP for 'Client
```

The evaluation of this involves checking the direction of the action and its arbitrary string.

- An asset transfer action is one that specifies an asset transfer operation from one party to another. This type of action is traceable which ensures that asset values for involved parties are updated accordingly. A transfer action is written as follows:

```
transfer 100 SAR by 'Tenant for 'Landlord
transfer 1000 USD by 'Client for 'ISP
```

In the first action, 100 SAR are transferred from the tenant to the landlord, while in the second one 1000 USD are transferred from the Client to the ISP.

3.2.7 Contract Body

The contract body contains the list of clauses that make up the contract. In a traditional contract, these clauses might appear governed by or free from conditions. In SlanC, conditional clauses map to SlanC rules, while unconditional ones

map to either agreements or containers.

3.2.8 Agreements

A SlanC agreement can be one of the following:

- Obligations, which can be written as follows:

```
[Party1] is obligated to [Action] for [Party2]
    [Time Constraint] [Time Reference Declaration]
```

- Permission, which can be written as follows:

```
[Party1] is permitted to [Action] for [Party2]
    [Time Constraint] [Time Reference Declaration]
```

```
[Party1] is not obligated to [Action] for [Party2]
    [Time Constraint] [Time Reference Declaration]
```

- Prohibition, which can be written as follows:

```
[Party1] is prohibited from [Action] for [Party2]
    [Time Constraint] [Time Reference Declaration]
```

The definition of each of these is inherited from deontic logic and extended to facilitate expressing additional information, such as time constraints. Furthermore, its worth noting that in these statements the writer is not conveying intent, rather, the convey the direction of the obligation, permission and prohibition, respectively. For example, in the case of an obligation it is understood that [Party1]

is the one who must perform the [Action] and [Party2] is the one who benefits/loses when its performed.

An agreement has two embedded components: a time reference declaration and a time constraint. They are defined as follows:

- Time Reference declaration: Allows the writer to define a string that reference the time the agreement is fulfilled. This reference can be used in the time constraints of other agreements or rules to build dependencies between this agreement and others. It is written as follows:

  ```
  where time is "fulfillment time reference"
  ```

- Time Constraint: Allow the contract writer to restrict the fulfillment of an agreement to a time frame. SlanC defines the following 13 types of constraints:

  1. **`at any time:`** This is fulfilled iff the event occurs at any time, i.e there is no constraint on time.
  2. **`at T`**: at a given time "T".
  3. **`before T`**: before a given time "T",
  4. **`on or before T`**: on or before a given time "T".
  5. **`after T`**: after a given time "T".
  6. **`on or after T`**: on or after a given time "T".
  7. **`between T1 and T2`**: between two given times "T1" and "T2" inclusive.

8. **`within OFFSET of T`**: in the period starting with the time “T” and ending with the time of “T” + “OFFSET” inclusive, where “OFFSET” is a time offset.

9. **`every OFFSET starting at T1 and ending at T2`**: The attached agreement is fulfilled iff the event described in the clause occurs exactly once at every ”OFFSET” between ”T1” and ”T2” starting at ”T1”.

10. **`at all times:`** Behaves exactly like the previous one, but this one is fulfilled even if the event occurs more than once at each offset.

11. **`at any time repeated:`** Just like the ”at any time” constraint, this one is fulfilled if the event occurs at least once, however, this one keeps track of all the other occurrences. Furthermore, the fulfillment of this one is asserted after the end of the contract.

12. **`at each [Time Reference]:`** The [Time Reference] refers to an event that repeatedly occurred many times. This constraint is only fulfilled iff each occurence of the event it is attached to has an adjacent occurence of the other event. For example, in the case of recurring events like paying rent every month and giving receipts for each of those payments, then this time constraint can be used to specify that there must be a receipt at each time there is a payment. More concrete examples on this and other time constraints will be presented later.

13. **`within OFFSET of each [Time Reference]:`** Behaves just like the previous one, but instead of having it at the same time as the other

event, it establishes a range. For example, the receipt in the previous example could be delivered within a week of when each payment is made.

Time references and time constraints are crucial for establishing dependencies between different agreements. We illustrate this through the following two agreements from the Internet Access Services contract:

```
'Client is obligated to "pay service fees" for 'ISP
  every (0 , 1 , 0) starting at (1 , 1 , 2018)
  and ending at (31 , 12 , 2018)
  where time is "fee payment time"
'ISP is obligated to "deliver payment receipt" for 'Client
  within (7 , 0 , 0) of each "fee payment time"
  where time is "receipt delivery time"
```

The first obligation states that the Client must pay the service fees at the beginning of every month starting from January and ending at December and the time for each payment is recorded in a time reference called fee payment time. This reference is used by the second obligation which states that the ISP must deliver a receipt for each payment within 7 days of that payment. The first agreement is breached if at least one of the payments is not made at the specified time. However, the second agreement can be fulfilled even if the first one is breached because the second one is tethered to the payments.

In general, the fulfillment of an obligation or a permission is defined as the

performance of the described action within the given time frame. The time reference refers to the time the action was performed. On the other hand, fulfillment of prohibitions is defined as the non performance of the specified action within the time frame. The time reference, in this case, refers to the time of the event that exceeds the highest time bound in the prohibitions time constraint.

Agreements must be preceded with an ID when they are written in the body of the contract. This ID is an arbitrary string and it can be anything, but the writer must ensure that it is unique within the contract. The agreement, then, can be written as follows:

```
"AG_001" ; [Party1] is obligated to [Action] for [Party2]
    [Time Constraint] [Time Reference Declaration]
"AG_002" ; [Party1] is permitted to [Action] for [Party2]
    [Time Constraint] [Time Reference Declaration]
"AG_003" ; [Party1] is prohibited from [Action] for [Party2]
    [Time Constraint] [Time Reference Declaration]
```

IDs simplify the process of identifying and referring to clauses in general and they have other uses which we discuss later.

3.2.9 Rules

A rule allows the contract writer to express conditional clauses. A rule is similar in structure to an if-then conditional. It has the following form:

```
if [Condition] [Time Reference Declaration] then
```

```
[Clause Container]
```

Rules allow the contract writer to check for the occurency or absence of events. Checking for absence of events is especially useful for expressing reparation clauses. In general, a rule is considered fulfilled, if its condition check is fulfilled and its consequent is fulfilled. Furthermore, a rule is composed from 3 main components: the condition, the consequent and the time reference declaration. The latter is expressed in the same way as in the agreements. The consequent is container which we discuss later. The condition is a boolean assertion about the occurrence or absence of a certain event. It can be written as follows:

```
[Action] by [Party1] for [Party2] [Time Constraint] is satisfied
[Action] by [Party1] for [Party2] [Time Constraint] is not
    ↪ satisfied
```

The first one asserts that an action was performed by [Party1] for [Party2] subject to the time constraint specified in [Time Constraint]. The second one asserts that the action was not performed subject to the time constraint.

Looking back at the internet access services contract, a rule in SlanC might look like the following:

```
if "decrease bandwidth usage" by 'Client for 'ISP is not satisfied
    before (31 , 1 , 2018) where time is "bandwidth usage decrease
        ↪ time"
then all {
    'Client is obligated to "pay \$50 fee" for 'ISP
```

```
        after (31 , 1 , 2018 ) where time is "fee pay time"
}
```

This rule states that if the client doesn't decrease bandwidth usage before the end of January then they must pay a $50 fee after the end of that month. Notice that the client will not be obligated to pay the fee if they decrease their bandwidth usage, and this is because the condition of the rule will only be true if they do not decrease it.

The consequent is a type of container, which we will discuss in detail the next section.

Rules, like agreements, must be preceded with an ID when they are written in the body of the contract. This ID is an arbitrary string and it can be anything, but the writer must ensure that it is unique. The rule, then, can be written as follows:

```
"Rule \#1" ; if [Condition] [Time Reference Declaration] then
        [Clause Container]
```

3.2.10 Containers

Clause container are another type of clauses that encapsulate a set of agreements and rules and other containers to add more semantic information to them. Currently, SlanC supports 4 types of containers:

- ALL: A conjunction container that is only fulfilled iff all the clauses contained within it are fulfilled. The all container is written as follows:

```
all where time is "Fulfillment time" {
    'Client is obligated to "pay \$50 fee" for 'ISP
        after (31 , 1 , 2018 ) where time is "fee pay time"
    'ISP is permitted to "deliver receipt" for 'Client
        within (14 , 0 , 0) of "fee pay time"
        where time is "fee pay time"
}
```

The all container will be fulfilled iff the client pays the fee and the ISP sends the receipt subject to their respective time constraints, otherwise the container is said to be breached.

- ANY: A disjunction container that is fulfilled iff at least one of the clauses within it is fulfilled. The any container is written as follows:

```
any where time is "Another Fulfillment time" {

'Tenant is obligated to "pay \$50 fee" for 'Landlord at
    ↪ (31 , 1 , 2018 ) where time is "fee pay time"

'Tenant is obligated to "pay \$100 fee" for 'Landlord
    ↪ after (31 , 1 , 2018 ) where time is "late fee pay
    ↪ time"
}
```

This gives the client the option to wither pay the fee on time or pay double its value any time later.

- PARALLEL: A conjunction container that is only fulfilled iff all the clauses contained within it are fulfilled by the same time. The parallel container is written as follows:

```
parallel where time is "a Fulfillment time" {
'Tenant is obligated to "pay rent" for 'Landlord
   after (31 , 1 , 2018 ) where time is "fee pay time"
'Tenant is obligated to "submit repair requests" for '
   ↪ Landlord
   after (31 , 1 , 2018 ) where time is "repair requests
      ↪ time"
}
```

This parallel container is fulfilled iff the client submits their repair request at the same time they pay their rent.

- SEQUENCE: A conjunction container that is fulfilled iff all the clauses contained within it are fulfilled one after the other in order of appearance within the container. The sequence container is written as follows:

```
sequence where time is "some fulfillment time" {
'Tenant is obligated to "pay rent" for 'Landlord
   after (31 , 1 , 2018 ) where time is "fee pay time"
```

```
'Tenant is obligated to "submit repair requests" for '
    ↪ Landlord
    after (31 , 1 , 2018 ) where time is "repair requests
        ↪ time"
}
```

This sequence container is fulfilled iff the client submits their repair requests after they pay their rent.

These containers allow the contract writer to establish interesting dependencies between clauses in general. There are two kinds of dependencies that we would like to highlight here:

- one-on-any dependencies established by using the `any` container, as follows:

```
any where time is "disj time" {
    'Tenant is obligated to "transfer rent" for 'Landlord
        before (29 , 1 , 2018) where time is "rent payment time
            ↪ "
    if "transfer rent" by 'Tenant for 'Landlord
        before (29 , 1 , 2018) is not satisfied
        where time is "no rent time"
    then
    {
        'Tenant is obligated to "notify for not being able to
```

```
            ↪ pay"
            for 'Landlord before (31 , 1 , 2018)
            where time is "notification time"
    }
}
'Landlord is prohibited from "evict" for 'Tenant
    within (0 , 1 , 0) of "disj time"
    where time is "no eviction time"
```

The prohibition about the eviction is time constrained by the time reference "disj time", which is defined by the disjunction container above, which in turn only defines that reference if the tenant pays the rent on time or sends a notification to the landlord telling him that he is not going to pay. In other words, the prohibition can only be fulfilled or breached iff one of the items in the disjunction container is fulfilled.

- one-on-many dependencies:

```
all where time is "conj time" {
    'Buyer is permitted to "order goods online" for 'Seller
    at any time where time is "order time"
    'Buyer is obligated to "pay the price" for 'Seller
    within (7 , 0 , 0) of "order time" where time is "pay time
        ↪ "
```

```
}
'Seller is obligated to "ship goods" for 'Buyer
within (7, 0 , 0) of "conj time" where time is "ship time"
```

The agreement where the seller ships the goods depends on ”conj time”, this reference is only defined when all the agreements within the conjunction container are fulfilled. This, in turn, ensures that the fulfillment and breach of the ”ship goods” obligation is only asserted after the container is fulfilled.

It is important to note that these containers behave differently from rules, in that their bodies are always executed even if the container is fulfilled or breached. This eliminates the need to repeat children of that container just because they existed within that container.

Containers like rules and agreements, must be preceded with an ID when they are written in the body of the contract. This ID is an arbitrary string and it can be anything, but the writer must ensure that it is unique. The container, then, can be written as follows:

```
"Container #1" ; [Container Specifier] [Time Reference
    ↪ Declaration] {
    [Clause List]
}
```

3.2.11 Contract Environment

The environment of the contract affects the clauses of the contract, either positively or negatively. In SlanC the environment is a time ordered list of events, where an event is a time stamped action and it is written as follows:

```
[Generic Action] @ [Time]

[Transfer Action] @ [Time] in fulfillment of [Clause ID]
```

3.2.12 Contract State

The state of the contract records all the information about all the clauses of the contract given the events that occurred in the environment. It contains the following 5 components:

- Asset List: a list of all the assets in that contract updated according to all the transfers that occurred in the environment.
- Time Reference List: A list of all the time references and their actual values. This is also referred to internally as the variable list.
- Clause State List: Records the final state of all clauses within the contract. A statement about 4 different variants and they are written as follows:

```
[Clause ID] fulfilled by [Time] with time recorded in [
    ↪ Time Reference]
[Clause ID] breached by [Time] with time recorded in [Time
    ↪  Reference]
```

```
[Clause ID] exercised by [Time] with time recorded in [
    ↪ Time Reference]
[Clause ID] not exercised by [Time] with time recorded in
    ↪ [Time Reference]
```

For example,

```
"Rule #1" fulfilled by (4 , 5 , 2018) with time recorded
    ↪ in "Rule #1 Time"
"AG1" breached by (6 ,7 , 2018) with time recorded in "AG1
    ↪  Time"
"EX11" exercised by (8 , 7 , 2018) with time recorded in "
    ↪ EX1"
"AGR2" not exercised by ( 9 , 9 , 2018) with time recorded
    ↪  in "AGR2 time"
```

The fulfilled/breached variants are used for all types of of clauses except permissions, where we use exercised/not exercised.

- Clause and Asset Pair List: This records the IDs of all the clauses that involve asset transfers and the remaining amount before fulfillment. It is written as follows:

```
[Clause ID] ; [Asset Quantity]
```

e.g.

```
"AGR1" ; 500
```

- Transaction List: This records all the transactions(asset transfers) that occurred with their timestamps. It is written as follows:

  ```
  transaction: [Event]
  ```

  For example,

  ```
  transaction: transfer 50 USD by 'Buyer for 'Seller
  ```

The state, then, can be expressed as follows:

```
[Asset List] | [Time Reference List] | [Clause State List] | [
    ↪ Clause and Asset Pair List] | [Transaction List]
```

3.2.13 Contracts and inheritance

In object-oriented programming, inheritance enables new objects to take on the properties of existing objects. A class that is used as the basis for inheritance is called a superclass or base class. A class that inherits from a superclass is called a subclass or derived class. The subclass can also override certain properties of the superclass.

The same concept is used here for contracts instead of classes. A contract can extend many other previously defined contracts and overriding some or all of their clauses while adding additional ones. This feature enables contract reuse through the use of simple and intuitive constructs. We anticipate that this will revolutionize the way contracts are drafted which in turn allows for better insight about contracts being changed between different drafts. For instance, in most

cases many businesses start from standard agreements that have been thoroughly vetted by their legal team, however, sometimes the alter them to accommodate certain partners/customers. Ding it the regular way of making a copy of the same contract and updating it or even modifying the original document, makes it very difficult use this piece of information about what was changed in the future.

The following example that illustrates how this feature is used in practice:

```
Contract{
    Preamble{
        name: "Parent"
        start time: (1 , 1 , 2018)
        end time: (1 , 12 , 2018)
        parties:
        [
            { id: 'a , name: "party a" , description: "A",
                address: noaddress}
            { id: 'b , name: "party b" , description: "B",
                address: noaddress}
        ]
        extends: none
      }
      Body {
        "1" ; 'a is obligated to "pay rent" for 'b
```

```
            at (31 , 1 , 2018) where time is "rent time"
        "2" ; 'a is obligated to "Say hello" for 'b
            at (1 , 5 , 2018) where time is "hello time"
      }
}
Contract
{
    Preamble {
        name: "Child"
        start time: (1 , 1 , 2018)
        end time: (1 , 12 , 2018)
        parties:
        [
            { id: 'a , name: "party a" , description: "A",
            address: noaddress}
            { id: 'b , name: "party b" , description: "B",
            address: noaddress}
        ]
        extends: "Parent"
    }
    Body {
        override "2" ; 'a is obligated to "say hi" for 'b
```

```
            at (1 , 7 , 2018) where time is "hi time"
    }
}
```

In this example the ”Child” contract extends the ”Parent” contract by including the name of the ”Parent” contracts in the extends field of its preamble. This enables SlanC to find all the agreements in ”Parent” and include them in the ”Child” at the time of the evaluation, without having to write them manually in the ”Child”. Furthermore, the ”Child” contract overrides an agreement from the parent which has ID ”2” through the use of the override construct. The `override` construct discards the clause with the given ID from the parent and includes the new one instead of it. Therefore, the resulting contract would be as follows:

```
Contract
{
    Preamble {
        name: "Child"
        start time: (1 , 1 , 2018)
        end time: (1 , 12 , 2018)
        parties:
        [
            { id: 'a , name: "party a" , description: "A",
            address: noaddress}
            { id: 'b , name: "party b" , description: "B",
```

```
            address: noaddress}
        ]
        extends: "Parent"
    }
    Body {
        "1" ; 'a is obligated to "pay rent" for 'b
            at (31 , 1 , 2018) where time is "rent time"
        "2" ; 'a is obligated to "say hi" for 'b
            at (1 , 7 , 2018) where time is "hi time"
    }
}
```

CHAPTER 4

FORMAL SPECIFICATION OF SLANC

One of the main features of SlanC is that it is readily formally definable. In this chapter we present the formalization of SlanC by presenting the the formal EBNF specification of its syntax (Section 4.1) and its semantics as developed in Maude (Section 4.2).

4.1 EBNF Specification of Syntax

In this section we present the EBNF specification of the syntax of SlanC.

⟨*Contract*⟩ ::= '`Contract{`' ⟨*Preamble*⟩ ⟨*Body*⟩ '`}`'

⟨*Preamble*⟩ ::= '`Preamble{`' '`name:`' ⟨*String*⟩ '`start time:`' ⟨*Time*⟩ '`end time:`'
⟨*Time*⟩ '`parties:`' ⟨*PartyList*⟩ '`extends:`' ⟨*StringList*⟩ '`}`'

⟨*Body*⟩ ::= '`Body{`' ⟨*ClauseList*⟩ '`}`' ⟨*ClauseList*⟩ ::= (⟨*Clause*⟩ | ⟨*OverrideClause*⟩)
{(⟨*Clause*⟩ | ⟨*OverrideClause*⟩)} | noclauses

⟨*OverrideClause*⟩ ::= '`override`' ⟨*Clause*⟩

⟨*Clause*⟩ ::= ⟨*ClauseID*⟩ '`;`' (⟨*Agreement*⟩ | ⟨*Rule*⟩ | ⟨*ClauseContainer*⟩)

⟨*Agreement*⟩ ::= ⟨*Obligation*⟩ | ⟨*Permission*⟩ | ⟨*Prohibition*⟩

⟨*Obligation*⟩ ::= ⟨*PartyID*⟩ '`is obligated to`' ⟨*Action*⟩ for ⟨*PartyID*⟩ ⟨*TimeConstraint*⟩
⟨*TimeReferenceDeclaration*⟩

⟨*Permission*⟩ ::= ⟨*PartyID*⟩ '`is permitted to`' ⟨*Action*⟩ for ⟨*PartyID*⟩ ⟨*TimeConstraint*⟩
⟨*TimeReferenceDeclaration*⟩

⟨*Prohibition*⟩ ::= ⟨*PartyID*⟩ '`is prohibited from`' ⟨*Action*⟩ for ⟨*PartyID*⟩ ⟨*TimeConstraint*⟩
⟨*TimeReferenceDeclaration*⟩

⟨*Rule*⟩ ::= '`if`' ⟨*Condition*⟩ ⟨*TimeReferenceDeclaration*⟩ '`then`' ⟨*BasicClauseContainer*⟩

⟨*Condition*⟩ ::= ⟨*Action*⟩ ⟨*TimeConstraint*⟩ ('`is satisfied`' | '`is not satisfied`')

⟨*BasicClauseContainer*⟩ ::= ⟨*ContainerSpecifier*⟩ '`{`' ⟨*ClauseList*⟩ '`}`'

⟨*ClauseContainer*⟩ ::= ⟨*ContainerSpecifier*⟩ ⟨*TimeReferneceDeclaration*⟩ '`{`' ⟨*ClauseList*⟩
'`}`'

⟨*ContainerSpecifier*⟩ ::= ('`all`' | '`any`' | '`parallel`' '`sequence`')

⟨*Time*⟩ ::= '`(`' ⟨*NaturalNumber*⟩ '`,`' ⟨*NaturalNumber*⟩ '`,`' ⟨*NaturalNumber*⟩ '`)`'

⟨*TimeReferenceDeclaration*⟩ ::= '`where time is`' ⟨*VariableName*⟩

⟨*TimeArithmetic*⟩ ::= (⟨*Time*⟩ ('`++`' | '`--`') ⟨*Time*⟩) | (⟨*VariableName*⟩ ('`+`' | '`-`')
⟨*Time*⟩)

⟨*VariableName*⟩ ::= ⟨*String*⟩ | ⟨*String*⟩ '`#`' ⟨*NaturalNumber*⟩

⟨*VariableValue*⟩ ::= ⟨*Time*⟩

⟨*Variable*⟩ ::= (⟨*VariableName*⟩ is ⟨*VariableValue*⟩) | ⟨*VariableGroup*⟩

⟨*VariableGroup*⟩ ::= ⟨*VariableName*⟩ '`group is {`' ⟨*VariableList*⟩ '`}`'

⟨*VariableList*⟩ ::= ⟨*Variable*⟩ {⟨*Variable*⟩} | novariables

⟨*PartyID*⟩ ::= '''⟨*UnicodeCharacter*⟩⟨*UnicodeCharacter*⟩

⟨*Party*⟩ ::= '`{`' '`id:`' ⟨*PartyID*⟩ '`,`' '`name:`' ⟨*String*⟩ '`,`' '`description:`' ⟨*String*⟩
'`,`' '`address:`' ⟨*String*⟩ '`}`'

⟨*PartyList*⟩ ::= ⟨*Party*⟩ {⟨*Party*⟩}

⟨*Asset*⟩ ::= '`{`' '`name:`' ⟨*String*⟩ '`,`' '`code:`' ⟨*AssetCode*⟩ '`,`' '`supply:`' ⟨*Integer*⟩ '`,`'
'`owner:`' ⟨*PartyID*⟩ '`}`'

⟨*AssetList*⟩ ::= ⟨*Asset*⟩ {⟨*Asset*⟩}

⟨*String*⟩ ::= '`"`'⟨*UnicodeCharacter*⟩'`"`'

⟨*StringList*⟩ ::= ⟨*String*⟩ {⟨*String*⟩}

⟨*AssetCode*⟩ ::= '`USD`' | '`EUR`' | '`SAR`'

4.2 Equational Maude Specification

While the informal semantics of SlanC give a clear a clear explanation of its behavior, developing formal semantics is crucial for precisely describing the meaning of a contract in the language. Furthermore, having a formal model of a contract is a prerequisite for mechanizing operations on contracts, such as conformance checking or violation detection .

In this section, we give a full description of the syntax and formal semantics of SlanC which we developed as an equational theory in order-sorted equational

logic [7], specified as a functional module in Maude [6]. The specification is executable in Maude, which means that we immediately obtain an interpreter for SlanC with which contracts can be simulated. Moreover, the specification leverages the arsenal of generic tools available with Maude, so that various analysis and verification techniques can be applied to contracts in SlanC. The following sections discuss all the functional modules that were developed. The full definition of all these modules is available in Appendix A.

4.2.1 Time Module

The time module defines the syntax and semantics of dealing with time. Time in SlanC is a 3-tuple that takes 3 natural numbers as parameters that represent the day, month and year, respectively. In Maude, we declare a new module called Time that extends NAT and BOOL, which are the modules that define natural numbers and booleans, respectively. Within this module we declare 3 sorts:

- Time: A sort that has the actual representation of time.
- TimeError: A sort that represents time errors, i.e invalid dates such as "0/3/2017".
- TimeOperation: A sort that represents time arithmatic operators, such as adding times and subtracting them.

After defining the sorts we make the Time module extend the TimeError module, because a valid Time is just a subset of all times which include invalid times, as follows:

```
fmod TIME is
  pr NAT .
  pr BOOL .
  sort Time TimeError TimeOperation .
  subsort Time < TimeError .
endfm
```

Then we define the syntax of time by declaring operators, as follows:

```
  op _,_,_ : Nat Nat Nat -> Time .
  op undefined-time : Time .
  op time-error : -> TimeError .
```

Time is is just a 3-tuple of natural numbers, while a TimeError is simply written as `time-error`. The (DD,MM,YYYY) is assumed when writing times, because the semantics of time arithmetic are based on it. Furthermore, the specification also supports undefined times through the `undefined-time` operator. Moreover, we also define time operators for adding, subtracting and comparing times, although presently the specification only supports adding times. These operators are defined as follows:

- (++) : Adding two times, e.g (1 , 1 , 2018) ++ (1 , 2 , 3) .
- (–) : Subtracting two times, e.g (1 , 1 , 2018) – (1 , 2 , 3)
- (+) : Adding a Time reference to a Time, e.g ”Payment time” + (1 , 2 , 3)

- (-) : subtracting a Time from a Time reference, e.g "Payment time" - (1 , 2 , 3)
- lte(T1,T2) : checks if T1 is less than or equal to T2 .
- lt(T1,T2) : checks if T1 is less than T2 .

The full definition of the time module is available in Appendix A

4.2.2 Parties Module

The parties module is a simple module that defines the syntax of defining a new party or lists of parties. A party has an ID, name, description and address. The ID belongs to the sort PartyID and name and description are strings, while the address belongs to the sort Address. Currently, PartyID extends the QID sort which allows writing a sequences of characters starting with the single quote ('), while Address extends the String sort just like name and description.

Furthermore, a list of parties is constructed by appending different parties and separating them with whitespace. A snippet of the module is shown below, while the full module is reserved for A.

```
op noaddress : -> Address .


op {id:_,name:_,description:_,address:_} :
    PartyID String String Address -> Party .
```

```
op noparties : -> PartyList .

op _ _ : PartyList PartyList

   -> PartyList [id: noparties assoc] .
```

4.2.3 Assets Module

The assets module is another simple module that defines how an asset how assets are written and asset lists can be constructed. An asset belongs to the sort Asset and it has a name, code, supply, and owner. The name is a simple string and the code belongs to the sort AssetCode, while the supply is an integer and the owner belongs to the sort PartyID. the code of the asset is used to identify that asset, and presently SlanC only supprts USD, SAR and EUR. Support for new types of assets can be easily added by adding more codes, e.g Yen or Pound etc.. . A snippet of this module is shown below, while the full module is reserved for A.

```
ops USD SAR EUR : -> AssetCode .

op {name:_,code:_,supply:_,owner:_} :

     String AssetCode Int PartyID -> Asset .

op noassets : -> AssetList .

op _ _ : AssetList AssetList

     -> AssetList [assoc id: noassets] .
```

4.2.4 Variables Module

Declaring and using time references is an important feature of SlanC, it allows expression of fairly complex dependency relationships between different clauses in a simple way. These time references are implemented as variables and variable groups in the Maude specification. The variables module defines all the syntax and the semantics pertaining to time references.

First we declare the following sorts:

```
sorts Variable VariableList VariableGroup VariableDeclaration
     VariableName VariableValue TimeArithmatic .
subsort String < VariableName < TimeArithmatic .
subsort Variable < VariableList .
subsort VariableGroup < Variable .
op _ # _ : String Nat -> VariableName .
op _ is _ : VariableName Time -> Variable .
op where time is _ : VariableName -> VariableDeclaration .
op _ group is {_} : VariableName VariableList -> VariableGroup .
op _ _ _ : VariableName TimeOperation Time -> TimeArithmatic .
op novariables : -> VariableList .
op _ _ : VariableList VariableList -> VariableList [assoc id:
   ↪ novariables] .
```

- VariableName: a sort that extends string which specifies how variable names are written.

- VariableValue: a generic sort that encapsulates the value of a variable. Currently, the value of a variable can only be of sort Time.

- VariableDeclaration: The time reference declaration statements, which is written as: `where time is "some string"`.

- Variable: A variable has one of 3 forms:

    1. Normal variable: a normal variable is written as `[Variable Name] is [Variable Value]`, e.g `"Payment time" is (1 , 1 , 2018)` .

    2. Sequence Variable: A sequence variable is just like a regular variable, but it has an extra sequence number after its name, as follows: `[Variable Name] # [Sequence Number] is [Variable Value]`, e.g `"Rent payment time" # 3 is (3 , 3 , 2018)`

    3. VariableGroup: Recall the time references that store the time of many occurrences, e.g "at all times", a variable group is used to store all those times, as follows: `[Variable Name] group is { [Variable List] }` , e.g `"Rent time" group is { "Rent time" # 2 is (2 , 2 , 2018) "Rent time" # 1 is (1 , 1 , 2018) }` .

- VariableList: A list of variables, written as follows: `[Variable] [Variable] [Variable]`, e.g `("first time" is (1 , 2 , 2018)) ("second time" is (3 , 3 , 2018)) ("Rent time" group is { "Rent time" # 2 is (2 , 2 , 2018) "Rent time" # 1 is (1 , 1 , 2018) } )`

- TimeArithmatic: A sort that defines time arithmetic operations on vari-

ables, as follows: `[Variable Name] [Time Operation] [Time]`, e.g `"Rent time" + (7 , 0 , 0)`

In addition to these constructs, generic function like constructs that are used to manipulate variables and expressions. Its important to note that these expressions are never used as part of writing a contract, they are used internally when evaluating contracts. The constructs that we define are as follows:

- `get-variable-single-time`: This construct gets the value of a variable from a list of variables, given the name of the variable and the list. This function will ignore any variable groups with the same variable name, it will only get the value of regular variables.
- `get-variable-group`: Finds a variable group from a variable list given the variable group name.
- `update-variable-group`: Given a variable name, a variable and a variable list, the function returns an updated variable list where the group with the given name is update with the given variable.
- `get-variable-list`: Given a variable group, this function return the variable list contained within that group.
- `count-variables`: Given a variable list, this function returns the number of elements in that list.
- `get-time-at`: Given a variable list and an index, this function returns the time stored in the variable at that index starting from the last element in

the list. This function is particularly useful for variable groups, because using this function the time of the nth fulfillment can be found. Counting starts from the last element because these lists behave like stacks, where the last element is put first.

- `heads-time`: This gets the first element in the list, i.e the last element that was added.

- `evaluate-time-arithmatic`: Given a time arithmetic expression that involves variables and a list of variables, this function attempts to find the value of a variable from the variable list and substitute it in expression and returning it.

- `get-variable-name`: Given a time arithmetic expression that involves variables, this function returns the variable name.

- `get-added-value`: Given a time arithmetic expression that involves variables, this function returns the value that is added or subtracted from the variable.

- `get-time-operation`: Given a time arithmetic expression that involves variables, this function returns the operation of that expression, i.e addition or subtraction.

The full module is also available in Appendix A.

4.2.5 Constraints Module

The Constraints module defines the syntax and semantics of time constraints. The time constraints we discuss in this section are exactly like the ones we discussed previous, except in this section we discuss them from the perspective of their implementation in Maude. Each of these time constraints have at least two variants, one where they are used with constant times and another where they are used with time references, as follows:

```
at (1 , 5, 2018)

at "payment time"

at "payment time" + (1 , 1 , 1 )

between (1 , 1 , 2018 ) and "payment time" + (7 , 0 , 0)

etc ....
```

These are defined in Maude using different operators for each form, as follows:

```
op at _ : Time -> TimeConstraint .

op at _ : TimeArithmatic -> TimeConstraint .
```

All of these time constraints are operators of sort TimeConstraint, which we declare and use to contain them. Each of these time constraints passes through 4 main functions that we declare and use to do all computations related to them. These functions are:

- `satisfies`: Given a time value, a variable name, a time constraint and a variable list, this function returns true if the given time satisfies the time

constraint. The variable name comes from the time reference declaration from the the clause that triggered a call to this function and it is sometimes needed to to assess satisfaction. The variable list is needed because sometimes time constraints have time references in them and these need to be evaluated before assessing satisfaction. For example, if the time constraint was `at (1 , 1 , 2018)` and the given time was `(2 , 1 , 2018)` then satisfies would return false, because the times do not match. The evaluation of most time constraints is a simple mapping of their meaning, however, there are 3 time constraints that are a bit more involved in their evaluation. These time constraints are :

- `satisfies(T4,VN,every T1 starting at T2 and ending at T3,VL)`: Check if T4 is equal to (Z + T1) , where Z is the time of the last occurrence in the group with variable name equals to VN. This holds because T1 is an offset, so if the time of the previous occurrence plus the offset is equal to the time of the current occurrence (T4), then this time constraint is satisfied.
- `satisfies(T4,VN,at all times,VL)`: always returns true .
- `satisfies(T4,VN,at any time,VL)`: always returns true .
- `satisfies(T4,VN,at any time repeated,VL)`: always returns true .
- `satisfies(satisfies(T4,VN1,at each VN2,VL))`: Check if (Z == Y + 1) and T4 == X , where Z and Y are the sizes of the groups defined by VN2 and VN1, respectively and X is the time of the last

occurence recorded in the variable group defined by VN2.

  - `satisfies(satisfies(T4,VN1,within T2 of each VN2,VL))`: Just like the previous one, however, instead of equating the time, it establishes a range from X to X + T2 and does the check based on that.

- `exceeds`: Given a time, time constraint and variable list, this function decides if the given time exceeds the highest bound in the time constraint.

- `update-with-constraint`: Given a variable name, time, time constraint and a variable list, this function returns an updated variable list that contains new variables. For most of these time constraints a new variable is added to the list with the given variable name and the given time. However, for some of these time constraints the update operations is a bit more involved, as follows:

  - `update-with-constraint(VN1, T1,at any time repeated,VL)`: This will always add a new variable to the group defined by VN1 containing a new variable in the correct sequence number with value equal to T1.

  - `update-with-constraint(VN1, T1,every T2 starting at T3 and ending at T4,VL)`: This will add a new variable to the group defined by VN1 containing a new variable in the correct sequence number with value equal to T1.

  - `update-with-constraint(VN1, T1,at all times,VL)`: This will add a new variable to the group defined by VN1 containing a new variable

in the correct sequence number with value equal to T1.

  - `update-with-constraint(VN1, T1,at each VN2,VL)`: This will add a new variable to the group defined by VN1 containing a new variable in the correct sequence number with value equal to T1, where the correct sequence number is obtained by counting the number of variables in the group defined by VN2 .

  - `update-with-constraint(VN1, T1,within T2 of each VN2,VL)`: Equivalent to the previous one.

  - `update-with-constraint(VN1, T1,within T2 of each VN2,VL)`: Equivalent to the previous one.

- `count-increments`: Given a time constraint and a varibale list, this function counts the number of time increments in a given time constraints. This returns 1 for all time constraints except the following ones:

  - `every T1 starting at T2 and ending at T3`: how many times can T1 be added to T2 before reaching T3.

  - `at all times`: how many times can (1 , 0 , 0) be added to "Start Time" before reaching "End Time". Since these two times are the beginning and end times of the contract, this is equivalent asking what is the length of the contract in days?.

There are also other small helper functions that we do not discuss here and we leave to the reader to look them up in Appendix A .

4.2.6 Actions Module

This module defines the syntax and semantics of actions. In this module 3 sorts are declared for actions, as follows:

- Action: The generic sort that defines all actions.
- DirectedBasicAction: Extends the generic action and adds the ability to express directed generic actions that are defined through strings. An action of this forms is written as : `[GenericAct] by [PartyID] for [PartyID]`, e.g `"Deliver goods" by 'Seller for 'Buyer` .
- DirectedAssetTransferAction: This also extends the generic action but it allows expressing directed asset transfer actions instead of generic string actions. It is written as follows: `[TransferAct] by [PartyID] for [PartyID]`, e.g `transfer 100 EUR by 'Buyer for 'Seller`.

We also define functions in this module that compute the effect of actions on assets. These functions are:

- `update-asset-values-with-action`: Given an action and a list of assets, this function updates the assets with the effect of the action. If the action is not a transfer action then the same asset list is returned, otherwise the transfer value is subtracted from the party that transferred, and added to the party that recieved. e.g transfer 100 USD by 'Buyer to 'Seller, will result in 100 USD being transferred from the 'Buyer to the 'Seller.

- `subtract-from-first-party`: given an asset list and a transfer action, this function returns an updated list where the amount of transfer in the action is subtracted from the party whose id is given first.

- `add-to-second-party`: Given an asset list and a transfer action, this function returns an updated list where the amount of transfer in the action is added to the party whose id is given second.

4.2.7 Conditions Module

This module defines the syntax for conditions. A Condition sort is declared, which extends the Bool sort. We define to operations on this sort that represent the two variants of conditions, as follows:

```
op __ is satisfied : Action TimeConstraint -> Condition .
op __ is not satisfied : Action TimeConstraint -> Condition .
```

These two variants define conditions that check for occurrence and absence of events.

4.2.8 Clauses Module

The clauses module declares the Clause sort, which is a generic sort that all types of clauses(agreements/rules/containers) derive from. It also declares another sort called OverrideClause, which defines clauses that are preceded by the `override` keyword. Furthermore, this module also defines the syntax of lists of clauses, which are white space separated concatenation of clauses.

4.2.9 Containers Module

The containers module defines the syntax used for expressing clause containers. Clause containers extend Clauses and there are two variants of containers that can appear in a contract, namely, the one that appears as an independent clause with its own id and the one that appears attached to a rule. Semantically, the two variants are evaluated in the same manner, in fact, rules construct a new independent container on the fly for the containers attached to them. This module implements all containers discussed in section 3.2.10.

4.2.10 Agreements Module

The agreements module defines the syntax for writing agreements. It declares 4 new sorts:

- Agreement: A generic sort that all types of agreements extend.
- Obligation: Extends the agreements sort and enables expressing obligations.
- Prohibition: Extends the agreements sort and enables expressing prohibitions.
- Permission: Extends the agreements sort and enables expressing permissions.

4.2.11 Rules Module

This module defines the syntax of writing rules. It declares a new sort called Rule

which allows expressing rules as shown in section 3.2.9. is available in Appendix A.

4.2.12 Preamble Module

This module defines the syntax of writing the contract preamble. It declares a new sort called Premable which allows expressing the preamble as shown in section 3.2.2. Also, this modules defines getter functions for the several preamble properties, as follows:

- get-name-field() : Gets the contract name field from the preamble.
- get-extends-field() : Gets the extends field.
- get-start-time-field() : Gets the start time field.
- get-end-time-field() : Gets the end time field.

4.2.13 Contract Module

This module defines the syntax of expressing a new contract. It declares two new sorts Contract and Body which represents the contract and the body, respectively. In addition, this module defines two convenience getter functions of certain fields from the preamble of the contract, as follows:

- `get-contract-name`: Gets the name of the contract from its preamble. This function uses the `get-name-field` function in the preamble module.

- `get-contract-ancestors-names`: Gets the list of all the names of contracts that the given contract extends. This function uses the `get-extends-field` function in the preamble module.

4.2.14 Environment Module

This module defines the syntax for expressing the environment of the contract, which is a time ordered list of events. Two sorts are declared, namely, Event and Environment. The Environment is the list of events concatenated with whitespaces. In addition this module declares the sorts Transaction and TransactionList for expressing transactions. A transaction is an event with an asset transfer action.

This module, also defines two convenience functions, as follows:

- `get-event-time`: Given an event, this function returns its time.
- `get-event-action`: Given an event, this function returns its action.

4.2.15 State Module

This module defines the contract state according to the specification in 3.2.12. It declares the following sorts:

- ClauseState: A sort that represents a statement about the fulfillment or breach of a clause.
- ClauseStateType: A sort that represents the type of the state of a given clause, there are two types: `fulfillment-state` and `breach-state` which

represent the state that is a fulfillment state and a state that is a breach state, respectively.

- GlobalClauseState: A sort that represents a list of clause states.
- ClauseAssetPair: A sort that represents a single clause asset pair.
- ClauseAssetPairList: A sort that represents a list of clause asset pairs.
- ContractState: A sort that represents the state of a contract. It is written as follows:

```
[AssetList] | [VariableList] |
[GlobalClauseState] | [ClauseAssetPairList] | [
    ↪ TransactionList]
```

For example,

```
{name:"Saudi Riyal" , code: SAR
, supply: 1000000 , owner: 'Landlord } |
"MyTimeReference" is (1 , 1 , 2018) |
"MyClauseID" fulfilled by (1 , 2 , 2018) with time
    ↪ recorded in "MyClauseIDTime" | "MyClauseID2" ; 500 |
transaction: transfer 50 USD by 'LandLord for 'Tenant
@ (1 , 1 , 2018 ) in fulfillment of "MyClauseID2"
```

In addition to defining the syntax for expressing the state of a contract, this module defines several helper functions, as follows:

- `find-state-of`: Given the ID of a clause and the global cluase state, this function will find the clause state of the clause with the given id.

- `find-state-of-by-vn`: Given the time reference of a clause(variable name) and the global clause state, This function finds the clause with that has the given time reference.

- `get-clause-state-type`: Given a clause state, this function returns either `fulfillment-state` or `breach-state` depending on the given clause state.

- `get-recorded-time`: Given a clause state, this function returns the fulfillment/breach time recorded in that clause state.

4.2.16 Core Function Module

This module defines several simple helper function that we use in defining the violation detection semantics. These functions are:

- `get-id`: Given a clause, this function returns its ID.

- `get-time-constraint`: Given a clause, this function returns its time constraint.

- `get-variable-name`: Given a clause, this function returns its time reference.

- `get-action`: Given a clause, this function returns the action described in that clause in the standard format: `[Act] by [PartyID] for [PartyID]`.

- `get-condition-id`: Given a clause ID, this function returns the same id concatenated with "-CONDITION". This function is included to simplify modifications to the semantics of rules later on, because it is used by the other functions that define the semantics of rules.

- `get-consequent-id`: Given a clause ID, this function returns the same id concatenated with "-CONSEQUENT". This function is included to simplify modifications to the semantics of rules later on, because it is used by the other functions that define the semantics of rules.

- `get-condition-vn`: Given a time reference(variable name), this function returns the same time reference without modifying it. This function is included to simplify modifications to the semantics of rules later on, because it is used by the other functions that define the semantics of rules.

- `get-consequent-vn`: Given a time reference(variable name), this function returns the same time reference concatenated with "-CONSEQUENT". This function is included to simplify modifications to the semantics of rules later on, because it is used by the other functions that define the semantics of rules.

4.2.17 Contract Extension Semantics Module

This module defines the semantics of contract inheritance, i.e how does a contract extend another contract. These semantics are defined through several functions, as follows:

- `filter-contract-list-by-names`: Given a list of contract names and a list of contracts, this function returns a new list of contracts that have their names in the names list.

- `extend-preamble`: Given two preambles `P1` and `P2`, this function extends `P1` with `P2` by including all the data from `P1` and adding all the parties from `P2` to `P1`.

- `resolve-override`: Given an override clause and a clause list, this function funds the clause to be overridden from the clause list and replaces it with the given clause.

- `resolve-overrides`: Given two lists of clauses `CSL1` and `CSL2`, this function calls the `resolve-override` function with every clause in `CSL1` against `CSL2`. This results in a clause list that contains all the clauses in `CSL2` updated with the overrides from `CSL1` and all the new clauses from `CSL1`.

- `extend-body`: Given two bodies `B1` and `B2`, this function extracts the clause lists from each body as `CSL1` and `CSL2`, respectively, then calls the function `resolve-overrides` with these two lists .

- `extend-single`: Given two contracts `C1` and `C2`, this function extends `C2` with `C1`, by calling `extend-preamble` and `extend-body` with the preambles and bodies of `C1` and `C2`, respectively.

- `extend-all`: Given a contract `C` and a filtered contract list `CL2` and a list of all contracts `CL`, this function calls it self with the following parameters:

  - The result of extending the contract `C` with from the first contract in CL2 by calling `extend-single` giving it the parameters `C` and `C2`, where `C2` is the results of resolving all the extensions of the first contract in the list `CL2` against `CL` using the `resolve-extensions` function.
  - a new version of `CL2` where the first contract is removed.
  - `CL`

- `resolve-extensions`: Given a contract `C` and a list of contracts `CL`, this function calls the function `extend-all` with the following parameters:
  - The contract `C`.
  - a new contract list `CL2` that was filtered against the names in the preamble of `C`.
  - `CL`.

Therefore, the general idea is to recursively resolve all the extensions of the contract and all its parents.

4.2.18 Violation Detection Semantics Module

This module defines the semantics of violation detection for contracts through a set of functions as follows:

- `detect-violations`: This is the entry point for detecting violations in a contract. It takes the contract, environment, asset list and a contract list and it produces the state of the contract after all the events in the environment

have occurred. It calls the function `compute-contract-state` with the following parameters:

  - The full version of the given contract after resolving its extensions using the function `resolve-extensions`.
  - The environment.
  - A new state that has the given asset list and empty lists for all other state components.
  -

- `compute-contract-state`: This function takes as input a contract, its environment and its initial state and it produces the final state after all the events have been processed. It calls the function `update-body-state-given-environment` with the following parameters:

  - The body of the given contract.
  - The environment.
  - The resulting state after updating the state of the contract with the preamble using the function `update-state-with` giving it the preamble of the contract and its initial state as parameters.

  compute-contract-state(ContractP B1,ENV,CS1) = update-body-state-given-environment(B1,ENV,update-state-with(P,CS1))

- `update-state-with`: This function takes the preamble of a contract and its

state as input and produces a new state that has its variable list updated to include the start and end times of that contract.

- `update-body-state-given-environment`: This function takes as input the body of the contract, its environment and initial state and it produces a new state after evaluating each event in the environment against all the clauses in the body. This function recursively calls its self with the following parameters:
  - The same body.
  - A new environment where the first event is removed.
  - The state obtained from the call to the function `update-body-state-given-event` with the following parameters:
    * The body of the contract.
    * The event that was removed from the environment.
    * The current state

  This composes the state as each event is evaluated against the body. Furthermore, this evaluation style is akin to real time evaluation, where the contract as a whole is evaluated against a single event each time. The implication of this is that within a real system the state can be saved in a database and when a new event occurs the saved state can be given as input to the function `update-body-state-given-event` along with the contract and and the event, and that would be equivalent to giving the entire envi-

ronment at once.

- `update-body-state-given-event`: This function takes as input a contract body, an event and an initial state and it produces the state after the body is evaluated against the event. This function calls `update-clauselist-state-given-event` with the following parameters:
  - The clause list extracted from the body.
  - The event.
  - An updated version of current state where the asset and transaction lists were updated as follow:
    * Asset list that has been updated with the effect of the action in the event by calling the function `update-asset-values-with-action` passing in the action in the event and the asset list as parameters.
    * A transaction list that has been updated by calling the function `update-transaction-list`
- `update-clauselist-state-given-event`: This function take a clause list, event and a state as input and produces a new state after the event has been evaluated against every clause in the clause list. It recursively calls itself with the following parameters:
  - The same clause list bu with the first event remove from it.
  - The event.

  - A new state obtained by evaluating the first clause from the clause list against the event in the current state by calling `update-clause-state-given-event` with the following parameters:
    * The clause.
    * The event.
    * the current state.

- `update-clause-state-given-event`: This function takes as input a clause, an event and a state and it produces a new state after evaluating the clause against the event in the current state. This function calls one of 3 functions depending on he clause type, as follows:
  - if the clause type is agreement it calls the function `update-with-agreement`, with the agreement, the event and the current state.
  - if the clause type is rule it calls the function `update-with-rule`, with the rule, the event and the current state.
  - if the clause type is container it calls the function `update-with-container`, with the container, the event and the current state.

- `update-with-agreement`: This function takes an agreement, an event and a state as input and it produces a new state after evaluating the agreement against the event in the current state. It calls the appropriate function depending on the specific type of the agreement, i.e `update-with-obligation`, `update-with-permission` and `update-with-prohibition`, for obligations,

permissions and prohibitions, respectively. All of these functions behave in exactly the same way, but we separate the evaluation of different components into different functions to simplify the process of introducing specifics to different components in the future. Each of these functions behaves as follows:

1. Check if the agreement has already been fulfilled or breached then just return the given contract state. This check is done through the functions `is-fulfilled` and `is-breached`, respectively. These functions require the clause id and the global clause state.
2. Check if the time in the event satisfies the time constraint in the agreement. This check is done by performing a call to the `satisfies` function which is defined in the constraints module and passing the following parameters:
   - The time of the event.
   - The time reference declared in agreement.
   - The time constraint of the agreement.
   - The variable list from the state.
3. check if the action in the event satisfies the action in the agreement through the function `is-action-satisfied`, which requires the clause id, the action and the event.
4. if both 1 and 2 are satisfied then return the state obtained by calling the function `check-and-update-fulfillment` with the following

parameters:

- The agreement.
- the time of the event.
- the result of updating the state with using the function using the function `update-assets-with-event` passing in the the clause id, the action of the agreement, the event and an updated contract state with the following components:
  * The same asset list component.
  * An updated variable list using the function `update-with-constraint-` which was declared in the constraints module- passing in the time reference declared in the agreement, the event time, the time constraint and the current variable list as parameters.
  * The current global clause state, clause asset pair and transaction lists.

5. if either 1 or 2 is not satisfied then the function `check-and-update-fulfillment` is called with the agreement, the event and the current state.

The call to `check-and-update-fulfillment` is done in both cases but with different parameters, to allow checking for breaches as we will see later.

- `update-with-rule`: This function takes a rule, an event and a state as input and it produces a new state after evaluating the rule against the event in the current state. This function behaves as follows:

1. Check if the condition and the consequent were fulfilled already, then return the same state. These checks are done through the function `is-fulfilled` passing in the ids as obtained from `get-condition-id` and `get-consequent-id` when giving them a clause id.

2. check if the condition was breached, then return the same state. This check is done through the function `is-breached` passing in the id as obtained from `get-condition-id` when giving it a clause id as a parameter.

3. check if the condition was fulfilled and the consequent was breached, then return the same state. These checks are done through the functions `is-fulfilled` and `is-breached` passing in the ids as obtained from `get-condition-id` and `get-consequent-id`, respectively when giving them the clause id.

4. check if only the condition was fulfilled then evaluate then evaluate the consequent against the event using the function `update-with-container` passing the following as parameters:

   - A new independent variant of the container is constructed by adding a clause id and a time reference declaration, where the clause id is equal to the result of calling `get-consequent-id` and the time reference equals to the result of calling `get-consequent-vn`
   - The event
   - the current state.

5. Check if the time in the event and the action satisfy the time constraint and the action in the condition in the rule, respectively, then the function `update-with-rule-special-case` is called with the following parameters:
   - The rule.
   - The event.
   - The new state obtained from calling the function `check-and-update-fulfillment` with the following parameters:
     * The result of calling `get-condition-id` on the id of the rule.
     * The condition.
     * The result of calling `get-condition-vn` on the time reference declared in the rule.
     * The time of the event.
     * The result of updating the state with using the function using the function `update-assets-with-event` passing in The result of calling `get-condition-id` on the id of the rule, the action in the condition, the event and an updated contract state with the following components:
       · The same asset list component.
       · An updated variable list using the function `update-with-constraint-` which was declared in the constraints module- passing in the the result of calling the function `get-condition-vn` on the

time reference declared in the rule, the event time, the time constraint and the current variable list as parameters.

- The current global clause state, clause asset pair and transaction lists.

6. If either one does not satisfy, then the function `check-and-update-fulfillment` is called with the result of calling `get-consequent-id` on the id of the rule, the event and the current state.

Notice that both types of rules, those that check for satisfaction and non satisfaction all evaluated the same without making any distinction between them. This is because all these differences are handled by the lower level functions `check-fulfilled` and `check-breached`.

- `update-with-rule-special`: This function has the same parameters as `update-with-rule` and it is to evaluate the consequent at the time the condition becomes fulfilled. The reason for this is that by the time we know the condition is fulfilled, we can no longer evaluate the consequent, therefore we use this function to do this special case check. If the condition is fulfilled then another call is performed to `update-with-rule` with the same parameters. Otherwise, the same contract is returned.

- `update-with-container`: This function takes as input a container, an event and a contract state and it produces a new contract state that represents the result of evaluating the container against the event in the current state. The function behaves as follows:

- if the container is already fulfilled or breached, then the function `update-clauselist-state-given-event` with the clause list contained within the container, the event and the contract state as parameters.
- if the container is not fulfilled or breached the the function `update-with-container-spe` is called with the following parameters:
  * The container.
  * The event.
  * the result of calling the function `update-clauselist-state-given-event` with the following parameters:
    · The list of clauses from the container.
    · The Event.
    · The current state.

- `update-with-container-special`: This function has the same parameters as the function `update-with-container` and this function behaves as follows:
  - It checks if the container is satisfied using the function `container-satisfied` giving it the container type specifier, the list of clauses contained in the container and the global clause list, then it returns the state resulting from calling the function `check-and-update-fulfillment` using the following parameters:
    * The container.

        * The event time.
        * An updated version of the current state where only the variable list has been updated using the function `update-with-constraint`, giving it as parameters the time reference, the event time ,"at any time" time constraint and the current variable list.
    - Otherwise, call the function `check-and-update-fulfillment` with the same parameters as above but without updating the state.

- `check-and-update-fulfillment`: This is an overloaded function that has two different variants that always produce a contract state:
    - The first variant is one that is used by all other clauses except rules. This one takes the following parameters:
        * The clause.
        * The event time.
        * The state of the contract.
    - The second variant is used by rules and it takes the following parameters:
        * Clause id, this ID will definitely contain "-CONDITION" because this variant is dedicated to checking conditions.
        * Condition
        * Time reference (variable name)
        * Event time

    * The state of the contract.

  Each of these variant will call the appropriate variant of `check-fulfilled` and `check-breached` to check for fulfillment or breach, respectively. Whichever call succeeds will result in an update to the global clause state of the contract with a statement about that clause. However, if the clause is a Rule then two statements are generated, one for the condition and the other for the consequent.

- `is-fulfilled`: This function takes as parameters a clause id and a global clause state and it produces boolean indicating if a clause is fulfilled(true) or not(false). It does so by checking the global clause state for fulfillment statements about the clause whose id is given.

- `is-breached`: This function takes as parameters a clause id and a global clause state and it produces boolean indicating if a clause is breached(true) or not(false). It does so by checking the global clause state for breach statements about the clause whose id is given.

- `container-satisfied`: This function takes as input a container type specifier, a clause list and a global clause state and it returns true or false depending on whether the container was satisfied or not, respectively. There are 4 types of containers, namely, all, any, parallel, sequence and this function checks the fulfillment based on the type of container, e.g the parallel container is fulfilled iff all the clauses are fulfilled at the same time etc...

- `check-fulfilled`: This function checks if a clause or a condition has been fulfilled or not in the current state, returning true or false, respectively. There are two variants(overloads) of this function, one that is used for rules and the other used for other types of clauses, as follows:
  - For rules: takes as input the condition of the rule, the time reference, the event time and contract state. This function behaves as follows:
    * if the condition checks for satisfaction, then the result is the logical and between the result of calling the functions `is-variable-count-match` and `is-tc-dependency-fulfilled`, with their respective parameters. This ensures that the number of occurrences of the events match the number described by the time constraint.
    * if the condition checks for non satisfaction, then the result is simply the negation of the previous one.
  - For other clauses: takes as input the clause, the event time and the contract state. This function behaves as follows:
    * For containers, obligations and permissions it is exactly the same as the first case in the rule variant.
    * For prohibitions, it is the logical and of the result of calling the functions `exceeds` and the negation of the result of `check-breached`. In other words, a prohibition is fulfilled if the time constraint is exceeded and this prohibition was not breached.
- `check-breached`: This function checks if a clause or a condition has been

fulfilled or not in the current state, returning true or false, respectively. This function, like `check-fulfilled` has the same two variants, however, evaluation is exchanged, as follows:

  - For rules, the behavior is as follows:
    * if the condition checks for satisfaction, then the result is the logical and between the result of calling the functions `exceeds` and the negation of `check-fulfilled`, with their respective parameters. This ensures that an agreement is only breached if its time constraint is exceeded and it hasn't been fulfilled.
    * if the condition checks for non satisfaction, then the result is simply the negation of the previous one.
  - For other clauses, the behavior is as follows:
    * For containers, obligations and permissions it is exactly the same as the first case in the rule variant.
    * For prohibitions, it is the logical and of the result of calling the functions `is-variable-count-matched` and the negation of the result of `is-tc-dependency-fulfilled`. In other words, a prohibition is breached if its action action has not been performed subject to its time constraint.

- `is-variable-count-match`: This function takes as input a time reference, a time constraint and a variable list and it produces a boolean value. The value is true if the number of variables in the variable list for the given time

reference equals the number of increments in the time constraint, which are obtained from `count-variables` and `count-increments`, respectively.

- `is-tc-dependency-fulfilled`: For time constraints that introduce dependencies between their clause and other clauses, this ensures an assessment about the current clause is not made until after the clause they depend on is fulfilled/breached. It takes as input an event time, a time constraint and a contract state and it produces a boolean value of either true (dependency fulfilled) or false (dependency not fulfilled yet). This function is especially used for specific type of time constraints, namely, at each [time reference], within [Time] of [Time reference] and at any time repeated. For the first two it checks if the clause defined by the time reference is fulfilled/breached, while for the last one it checks if the current event time exceeds the end time of he contract.

- `update-assets-with-event`: This function takes as input, a clause id, the action of the clause, the event, the time constraint and the state and it produces a new state where the asset clause pairs list has been updated with the effect of the event. It does so ensuring that the actions of the event and the clause match and that they are transfer actions and then it calls the function `update-asset-clause-pairs` with the following parameters:
  - The clause id.
  - The action of the clause.
  - The time constraint.

  - The variable list from the state.
  - The clause and asset pair list from the state.

  Which results in an updated asset clause pair list that replaces the one from the current state and then this final state is returned.

- `update-transaction-list`: Given a transaction list and an event, this function adds the event to the transaction list iff the event has an action of type asset transfer action.

- `update-asset-clause-pairs`: This function takes as input a clause id, a transfer action, a time constraint, a variable list and a clause asset pair list and it produces an updated asset clause pair list, where the remaining amount for that specific clause is reduced by the amount in the transfer action, if a record for that clause is found. Otherwise, a new record is created for that clause by using its id and giving it an amount that is computed by multiplying the number of times the transfer must occur-as observed in the time constraint- by the given amount in the action of the clause.

CHAPTER 5

VALIDATION OF THE EQUATIONAL SPECIFICATION

In this chapter we validate the equational specification of SlanC in Maude. We do this in 3 major ways:

- Basic test cases that were developed as we were developing the Maude specification, to do preliminary checks of correctness.
- Examples that were taken from the literature, to further increase our confidence in the conformance of the specification to what we had in mind while developing it and to check for conformance with other formalisms and specifications.
- Real world examples, to test the limits of the expressiveness of SlanC and increase confidence in it ever more.

We discuss each of those ways in depth in sections 5.1, 5.2 and 5.3, respectively. Furthermore, we present the threats to the validity of those results in section 5.4.

5.1 Validation through basic test cases

As the specification of SlanC was being developed, many unrealistic test cases were developed to do basic correctness checking. These test cases cover are categorized by module, so each module has its test cases separated into a single file. All of the developed test cases are available in Appendix B, but here we show the types of tests that we conducted through basic examples.

We developed two kinds of tests, namely, parsing tests and reduction tests. The parsing tests ensure that the constructs of SlanC parse (validate the structure) without any problems with the Maude parser. The result of a successful parse operation is the identification of the sort of the parsed construct. While reduction tests validate equations by evaluating those function like constructs which define the semantics of contract violation, and check the actual results against expected ones.

An example of a parsing test is the following:

Listing 5.1: Example of a Parsing Test

```
parse { name: "Saudi Riyals", code: SAR , supply: 500000, owner: '
   ↪ PARTY } .
--- which produces:
Asset: { name: "Saudi Riyals", code: SAR , supply: 500000, owner: '
```

```
↪ PARTY }
```

Since Maude was able to identify the type of the construct as an asset then we know that the statement was successfully parsed.

An example of a reduction test is the following:

Listing 5.2: Example of a Reduction Test

```
reduce get-single-variable-time-value("xvc",
                    ("xvc" is (15 , 5 , 5))
                    ("XBC" group is {
                        ("xvc" is (15 , 5 , 5))
                        ("xvc" # 5 is (15 , 5 , 5))
                        })).
--- which produces:
Time: (15 , 5 , 5)
```

The expected outcome of this reduction operation is the value of the variable called ”xvc” which is (15 , 5 , 5) and the returned result is equal to that. We performed several tests like this one for the different constructs in the language to validate their correctness under varying circumstances.

5.2 Validation through literature examples

In this section we validate SlanC through different examples we gathered from other papers that presented formalisms for contracts.

5.2.1 American Call Option

The American call option gives the buyer the option to buy one share of stock at some time in the future from the seller [1]. It is defined as follows:

1. If the buyer buys the option, which expires on January 1st, 2018, to purchase one share of stock by transferring $5 on June 1st 2018 then he can exercise that option at any time in that period, inclusive. The strike price of the option is $80.

2. If the buyer chooses to exercise that option by paying the strike price on December 1st 2018, then the seller must transfer one share of stock to the buyer within 30 days of the payment date.

Section 5.2.1.1 presents the specification of this contract in SlanC and section 5.2.1.2 presents its simulation and results using the semantics developed in Maude.

5.2.1.1 Specification in SlanC

These are typical clauses that are normally found in stock trading contracts. This example can be translated into the SlanC contract shown in Figure 5.1.

This example highlights the following features of SlanC:

- **structure and content:** The SlanC specification maintains most of the overall structure and content from the original contract, which makes it more maintainable.

- **User definable events:** The events described in the agreements and rules are user definable arbitrary strings.

Listing 5.3: Specification of the American Call Option Contract in SlanC

```
Contract
{
  Preamble {
    name: "American Call Option Contract"
    start time: (1 , 1 , 2018)
    end time: (1 , 12 , 2018)
    parties: [{ id: 'Buyer , name: "Stock Buyer"
                ,description: "A buyer" , address: "123 st." }
              {id: 'Seller , name: "Stock Seller"
                ,description: "A seller" , address: "456 st."}
    extends: none ]
  }
  Body
  {
    "Option buy" ; if transfer 5 USD by 'Buyer for 'Seller
       at "Start Time" is satisfied
       where time is "buy time"
    then all
    {
       "Exercise option" ; if transfer 80 USD by 'Buyer for 'Seller
          between "buy time" and "End Time"
          is satisfied
          where time is "exercise time"
       then all
       {
          "Stock transfer" ; "Seller" is obligated to transfer 1 SAR for
             ↪  'Buyer
             within (30 , 0 , 0) of "exercise time"
             where time is "stock transfer time"
       }
    }
  }
}
```

Figure 5.1: The American call option contract in SlanC

- **Simple dependency between clauses:** SlanC allows creating dependencies between different rules and agreements and sequencing them in a way that closely resembles natural language, through the use for time references. Time references are naturally declared (e.g where time is ”buy time”) and used (e.g between ”buy time” and ”end time”).

5.2.1.2 Simulation in the Maude

We simulated this contract in 3 different environments, one were the agreement was fulfilled, another were the agreement is breached and a third were the condition of the inner rule is not satisfied.

Case where the agreement is fulfilled: In this case we want the agreement to be fulfilled, but in order for that to happen the conditions of the parent rule and its parent must be fulfilled first. That is for this agreement to be fulfilled, then there must be at least 3 events in the environment that satisfy the conditions and the agreement, respectively. There are many environments that can potentially satisfy the agreement and this is due to the fact that the time constraints of the condition of the inner rule and the agreement use ranges of times, so any time within those ranges would lead to a fulfilled agreement. One such environment is shown in Listing 5.4.

Listing 5.4: Environment of case where the agreement is fulfilled

```
(transfer 5 USD by 'Buyer for 'Seller @ (1 , 1 , 2018)
   0in fulfillment of "Option buy")
```

```
(transfer 80 USD by 'Buyer for 'Seller @ (1 , 2 , 2018)
    in fulfillment of "Exercise option")
(transfer 1 SAR by 'Seller for 'Buyer @ (8 , 2 , 2018)
    in fulfillment of "Stock transfer")
("Thanks" by 'Buyer for 'Seller @ (9 , 2 , 2018) )
```

Therefore we evaluate the contract in SlanC by using the function `detect-violations` passing in the contract, the aforementioned environment and the asset list presented in 5.5.

Listing 5.5: Environment of case where the agreement is fulfilled

```
{ name: "Stocks", code: SAR , supply: 1000, owner: 'Seller }
{ name: "United States Dollars", code: USD , supply: 1500, owner: '
    ↪ Buyer }
```

This would result of the simulation is shown in Figure 5.2, which contains the following elements:

- Asset list [Shown in red]: The given asset list is updated after giving with the effects of the event after all the events occurred.
- Time references or variables [Shown in green]: Each time reference and its recorded value.
- Global clause state [Shown in white and green]: The status of each clause
- Clause asset pair list [Shown in yellow]: The remaining amount to transfer to fulfill this agreement.

- Transaction list [Shown in blue]: a list of all transfers that occurred in the environment.

There are a couple of important points to note about this result:

- The state contains separate entries about the conditions and consequents of a given rule, which are suffixed with ”-CONDITION” and ”-CONSEQUENT”, respectively. Except for variable names, where conditions inherit the variable name of the rule, while consequents get suffixed.

- The consequent of a rule is only evaluated after its condition is fulfilled.

```
result ContractState: ({name: "Stocks",code: SAR,supply: 999,owner: 'Seller} {name:
    "United States Dollars",code: USD,supply: 1415,owner: 'Buyer}) | (
    "buy time-CONSEQUENT" is 9,2,2018) ("exercise time-CONSEQUENT" is 8,2,2018) (
    "stock transfer time" is 8,2,2018) ("exercise time" is 1,2,2018) ("buy time" is 1,
    1,2018) ("Start Time" is 1,1,2018) "End Time" is 1,12,2018 | (
    "Option buy-CONSEQUENT" fulfilled by 9,2,2018 with time recorded in
    "buy time-CONSEQUENT") ("Exercise option" fulfilled by 9,2,2018 with time recorded
    in "exercise time") ("Exercise option-CONSEQUENT" fulfilled by 8,2,2018 with time
    recorded in "exercise time-CONSEQUENT") ("Stock transfer" fulfilled by 8,2,2018
    with time recorded in "stock transfer time") ("Exercise option-CONDITION" fulfilled
    by 1,2,2018 with time recorded in "exercise time") "Option buy-CONDITION" fulfilled
    by 1,1,2018 with time recorded in "buy time" | ("Option buy-CONDITION" ; 0) (
    "Exercise option-CONDITION" ; 0) "Stock transfer" ; 0 | transaction ; (transfer 1
    SAR by 'Seller for 'Buyer @ 8,2,2018 in fulfillment of "Stock transfer")
    transaction ; (transfer 80 USD by 'Buyer for 'Seller @ 1,2,2018 in fulfillment of
    "Exercise option") transaction ; (transfer 5 USD by 'Buyer for 'Seller @ 1,1,2018
    in fulfillment of "Option buy")
```

Figure 5.2: Case where the agreement is fulfilled simulation

Case where the agreement is breached In this case we want the agreement to be breached. Just like in the case of fulfillment, the conditions of parent rules have to be fulfilled, so the events that trigger them are lift as si, while the event that triggers the agreement is removed as shown in the environment in Listing **??**:

Listing 5.6: Environment of case where the agreement is breached

```
(transfer 5 USD by 'Buyer for 'Seller @ (1 , 1 , 2018)
   in fulfillment of "Option buy")
(transfer 80 USD by 'Buyer for 'Seller @ (1 , 2 , 2018)
   in fulfillment of "Exercise option")
(transfer 1 SAR by 'Seller for 'Buyer @ (10 , 3 , 2018)
   in fulfillment of "Stock transfer")
("The stock was transferred late!" by 'Buyer for 'Seller
      @ (10 , 3 , 2018) )
```

Evaluating the contract with the same asset list in Case 1 would result of the simulation is shown in Figure 5.3, which has a breached agreement which makes the inner consequent breached, which in turn makes the outer consequent breached. This holds because the two consequents are containers of type ”all” which are only fulfilled if all contained clauses are fulfilled, otherwise they are only breached if at least a single contained clause is breached.

Case where the condition of the inner rule is not satisfied: In this case we remove only the event that satisfies the inner rule, leaving the rest of the events in tact to see the effect of a rules condition. Thus, the environment passed during the evaluation becomes:

```
(transfer 5 USD by 'Buyer for 'Seller @ (1 , 1 , 2018)
   in fulfillment of "Option buy")
```

```
result ContractState: ({name: "Stocks",code: SAR,supply: 999,owner: 'Seller} {name:
    "United States Dollars",code: USD,supply: 1415,owner: 'Buyer}) | ("exercise time"
    is 1,2,2018) ("buy time" is 1,1,2018) ("Start Time" is 1,1,2018) "End Time" is 1,
    12,2018 | ("Option buy-CONSEQUENT" breached by 10,3,2018 with time recorded in
    "buy time-CONSEQUENT") ("Exercise option" breached by 10,3,2018 with time recorded
    in "exercise time") ("Exercise option-CONSEQUENT" breached by 10,3,2018 with time
    recorded in "exercise time-CONSEQUENT") ("Stock transfer" breached by 10,3,2018
    with time recorded in "stock transfer time") ("Exercise option-CONDITION" fulfilled
    by 1,2,2018 with time recorded in "exercise time") "Option buy-CONDITION" fulfilled
    by 1,1,2018 with time recorded in "buy time" | ("Option buy-CONDITION" ; 0)
    "Exercise option-CONDITION" ; 0 | transaction ; (transfer 1 SAR by 'Seller for
    'Buyer @ 10,3,2018 in fulfillment of "Stock transfer") transaction ; (transfer 80
    USD by 'Buyer for 'Seller @ 1,2,2018 in fulfillment of "Exercise option")
    transaction ; (transfer 5 USD by 'Buyer for 'Seller @ 1,1,2018 in fulfillment of
    "Option buy")
```

Figure 5.3: Case where the agreement is breached simulation

```
(transfer 1 SAR by 'Seller for 'Buyer @ (10 , 3 , 2018)
    in fulfillment of "Stock transfer")
("A dummy event!!" by 'Buyer for 'Seller
       @ (10 , 3 , 2018) )
```

Evaluating the contract with the same asset list in Case 1 would result of the simulation is shown in Figure 5.4

```
result ContractState: ({name: "Stocks",code: SAR,supply: 999,owner: 'Seller} {name:
    "United States Dollars",code: USD,supply: 1495,owner: 'Buyer}) | ("buy time" is 1,
    1,2018) ("Start Time" is 1,1,2018) "End Time" is 1,12,2018 | "Option buy-CONDITION"
    fulfilled by 1,1,2018 with time recorded in "buy time" | "Option buy-CONDITION" ; 0
    | transaction ; (transfer 1 SAR by 'Seller for 'Buyer @ 10,3,2018 in fulfillment of
    "Stock transfer") transaction ; (transfer 5 USD by 'Buyer for 'Seller @ 1,1,2018 in
    fulfillment of "Option buy")
```

Figure 5.4: Case where the condition of the inner rule is not satisfied

5.2.2 Agreement to Provide Legal Services

In this section, we translate a slightly modified version of a typical agreement to provide legal services presented in [30] into SlanC and simulate it under different

circumstances. The agreement has the following sections:

1. The attorney shall provide legal services up to (n) hours per month, and furthermore provide services in excess of (n) hours.

2. The company shall pay a monthly fee of (amount in dollars) and (rate) per hour for any services in excess of (n) hours.

3. This contract is valid from 1/1/2017 to 30/4/2017.

Section 5.2.2.1 presents the specification of this contract in SlanC and section 5.2.2.2 presents its simulation and results using the semantics developed in Maude.

5.2.2.1 Specification in SlanC

The agreements can be translated as follows:

Listing 5.7: Specification of the Agreement to Provide Legal Services in SlanC

```
Contract {

    Preamble {

        name: "Legal services Contract"

        start time: (1 , 1 , 2018)

        end time: (1 , 5 , 2018)

        parties: [

                  { id: 'Attorney , name: "Some Lawyer"

                  ,description: "A Lawyer" , address: "123 st." }

                  {id: 'Client , name: "Some Client"
```

```
                ,description: "A person" , address: "456 st."}
            ]
    extends: none
}
Body{
    "Legal service hours" ; 'Attorney is obligated to "provide
        ↪ 10 legal service hours"
    for 'Client every (0 , 1 , 0)
    starting at (25 , 1 , 2018) and ending at (25 , 1 , 2018)
    where time is "Legal Service Time"
    "Payment for service hours" ; 'Client is obligated to
        ↪ transfer 100 USD
    for 'Attorney at each "Legal Service Time"
    where time is "Payment time"
    "An Excess service hour" ; 'Attorney is not obligated to "
        ↪ Extra hour of legal service"
    for 'Client at any time repeated
    where time is "Extra service hour time"
    "Excess service hour payment" ; 'Client is obligated to
        ↪ transfer 15 USD
    for 'Attorney at each "Extra service hour time"
    where time is "Extra service hour time"
```

```
    }
}
```

This example shows how time constraints with repetition such as ”every” and ”any time repeated” can be used to express fairly complex conditions concisely. A good example of such complex condition are the first and second obligations. These two obligation combined state that there must be a payment of 100 from the client to the attorney, for every 10 hours of legal services provided by the attorney per month. This entails that if the client is only obligated to transfer the money only for the months where the attorney provided service. Therefore, in some cases the second obligation might be fulfilled even though the first one is breached, and that is because of type of dependency created by the ”at each [Time Reference]” construct.

The third and forth agreements are interesting because the same type of dependency is created between a permission and an obligation. These two agreements combined state that the attorney is allowed give an extra hour of legal service, and for each one the client must give the attorney 15 dollars. The ”at any time repeated” is an interesting construct, because it allows tracking all the times an event was made to fulfill third agreement, which in turn triggers another fulfillment for the 4th one.

5.2.2.2 Simulation in Maude

We simulated this contract in different environments to illustrate the behavior of the constructs we discussed in the specification.

Case with a single arbitrary event after the end time of the contract: We choose this case to illustrate the behavior of the "at each [Time Reference]" construct. The results of the simulation in the following environment with no assets is shown in Figure **??**:

```
("A dummy event!!" by 'Attorney for 'Client
        @ (2 , 5 , 2018) )
```

```
result ContractState: noassets | ("Start Time" is 1,1,2018) "End Time" is
    1,5,2018 | ("Excess service hour payment" fulfilled by 2,5,2018 with
    time recorded in "Extra service hour time") ("An Excess service hour"
    not exercised by 2,5,2018 with time recorded in
    "Extra service hour time") ("Payment for service hours" fulfilled by 2,
    5,2018 with time recorded in "Payment time") "Legal service hours"
    breached by 2,5,2018 with time recorded in "Legal Service Time" |
    nopairs | notransactions
```

Figure 5.5: Case with single arbitrary event after the end time

The reason we have a single event after the end time of the contract is because this is the only way to tell the current time, because SlanC uses events for time keeping. Since the event is after the end of the contract, SlanC ensures that every single clause in the contract has a statement about it in the state because, all the events that come after the end time will not contribute to the clauses of the contract. The results from the figure show that the permission is not exercised and the first obligation is breached, while both obligations that use the "at each

[Time Reference]" construct are fulfilled. The reason for this is that the client should only pay for provided legal services, and since no services were provided then the client is not obligated to pay for anything, which is why they are fulfilled.

The difference between "breached" and "not exercised" is that the former represents all types of clauses except permissions which are represented by "exercised/not exercised".

Case where one of the agreements that use "at each [Time Reference] is breached: In this case we would like to illustrate a case where these agreements that use the "at each [Time Reference]" are breached. The most basic case where there is such a breach is modify the environment such that there are legal services provided and no payment is made. An environment that produces such a state is:

```
"provide 10 legal service hours" by 'Attorney for 'Client
   @ (25 , 1 , 2018)
transfer 100 USD by 'Client for 'Attorney
   @ (25 , 1 , 2018) in fulfillment of "Payment for service hours"
"provide 10 legal service hours" by 'Attorney for 'Client
   @ (25 , 2 , 2018)
transfer 100 USD by 'Client for 'Attorney
   @ (25 , 2 , 2018) in fulfillment of "Payment for service hours"
"provide 10 legal service hours" by 'Attorney for 'Client
   @ (25 , 3 , 2018)
transfer 100 USD by 'Client for 'Attorney
```

```
   @ (25 , 3 , 2018) in fulfillment of "Payment for service hours"
"Extra hour of legal service" by 'Attorney for 'Client
   @ (23 , 4 , 2018)
"provide 10 legal service hours" by 'Attorney for 'Client
   @ (25 , 4 , 2018)
transfer 100 USD by 'Client for 'Attorney
   @ (25 , 4 , 2018) in fulfillment of "Payment for service hours"
"Dummy event!!!" by 'Attorney for 'Client
   @ (2 , 5 , 2018)
```

Figure 5.6 shows the results of this simulation, where the obligation to pay for each excess hour is breached. Because there wan an excess hour of service provided by the attorney that the client did not pay for. On the other hand, the other obligations that uses the ”at each [Time Reference]” construct is fulfilled because the client paid for all the regular service hours provided by the attorney.

Notice that in this case and all previous cases, we included a dummy event that exceeds the time of the contract. This is because we would like to have a statement about each of the agreements of in the contract when after the simulation. This dummy event must be explicitly given by the user and it has the following meaning: What would be the state of the contract, given that nothing else happens until the end time is reached?. The event must be explicitly given because the user might choose that they just want to know the state of a contract up to a certain point

```
result ContractState: noassets | ("Start Time" is 1,1,2018) ("End Time" is 1,5,
    2018) ("Legal Service Time" group is{("Legal Service Time" # 3 is 25,3,2018) (
    "Legal Service Time" # 2 is 25,2,2018) "Legal Service Time" # 1 is 25,1,2018})
    ("Payment time" group is{("Payment time" # 3 is 25,3,2018) ("Payment time" # 2
    is 25,2,2018) "Payment time" # 1 is 25,1,2018}) "Extra service hour time"
    group is{"Extra service hour time" # 1 is 23,4,2018} | (
    "Excess service hour payment" breached by 2,5,2018 with time recorded in
    "Excess payment time") ("An Excess service hour" exercised by 2,5,2018 with
    time recorded in "Extra service hour time") ("Payment for service hours"
    fulfilled by 25,3,2018 with time recorded in "Payment time")
    "Legal service hours" fulfilled by 25,3,2018 with time recorded in
    "Legal Service Time" | "Payment for service hours" ; -200 | transaction ; (
    transfer 100 USD by 'Client for 'Attorney @ 25,4,2018 in fulfillment of
    "Payment for service hours") transaction ; (transfer 100 USD by 'Client for
    'Attorney @ 25,3,2018 in fulfillment of "Payment for service hours")
    transaction ; (transfer 100 USD by 'Client for 'Attorney @ 25,2,2018 in
    fulfillment of "Payment for service hours") transaction ; (transfer 100 USD by
    'Client for 'Attorney @ 25,1,2018 in fulfillment of
    "Payment for service hours")
```

Figure 5.6: Case where one of the agreements that use "at each [Time Reference] is breached

in time before its end, where some of the clauses might not have any statements about them in the state.

5.2.3 Non-disclosure Agreement

Non-disclosure agreements are widely used to set expectations and limitations when engaging in an activity. We present an example of a non-disclosure agreement adapted from [21]. The agreement has the following clauses:

1. This agreement is valid for 5 years starting from 1/1/2017.

2. The employee must not disclose any information about the work carried out for the employer.

Section 5.2.3.1 presents the specification of this contract in SlanC and section 5.2.3.2 presents its simulation and results using the semantics developed in Maude.

5.2.3.1 Specification in SlanC

The example is translated into SlanC as follows:

Listing 5.8: Specification of A Non Disclosure Agreement in SlanC

```
Contract {
    Preamble{
        name: "Non disclosure agreement"
        start time: (1 , 1 , 2018)
        end time: (31 , 12 , 2022)
        parties: [
                  { id: 'Employer , name: "Some employer"
                  ,description: "An employer" , address: "123 st."
                     ↪ }
                  {id: 'Employee , name: "Some Employee"
                  ,description: "An employee" , address: "456 st."}
                ]
        extends: none
    }
    Body{
        "Prohibition Clause" ; 'Employee is prohibited from
           "disclose info about the work done" for 'Employer at any
              ↪  time repeated
           where time is "NDA violation time"
```

```
    }
}
```

In this contract we showcase prohibitions, which are specified in a similar way to other types of agreements. This agreement also uses the ”at any time repeated” construct which allows recording of all the violations of this prohibition as we will see in the evaluation section.

5.2.3.2 Simulation in Maude

We simulate this example in 3 cases:

- One where no events are given.
- A single event is given after the end time.
- Multiple violation events are given.

Case where no events are given: We simulate the contract when nothing happened in the environment. The result is shown in 5.7.

```
result ContractState: noassets | ("Start Time" is 1,1,2018)
    "End Time" is 31,12,2022 | noglobalstate | nopairs |
    notransactions
```

Figure 5.7: Case where no events are given

The result is a state where only the start and end times are defined. This is to be expected, because giving no events is equivalent to saying that we are at the start of the contract and nothing has happened.

Case where the prohibition is fulfilled: We simulate the contract in an environment where the prohibition is fulfilled. Because the prohibition spans the duration of the contract, then it suffices to have a single arbitrary event after the end of the contract. We choose this event to be an event that triggers a violation, however since its outside the bounds defined by the ”at any time repeated” time constraint, then it will not trigger a violation. The result is shown in 5.8.

```
result ContractState: noassets | ("Start Time" is 1,1,2018)
    "End Time" is 31,12,2022 | "Prohibition Clause" fulfilled by
    1,1,2023 with time recorded in "NDA violation time" | nopairs
    | notransactions
```

Figure 5.8: Case where the prohibition is fulfilled

The result is a state similar to the previous case with one difference; a statement about the fulfillment of the prohibition. Furthermore, notice how there are no variables defined using the time reference of the prohibition because no event in the environment violated it.

Case where the prohibition is breached: We simulate the contract in an environment where the prohibition is breached. Because the prohibition spans the duration of the contract, then it suffices to have a single disclosure event that violates the prohibition. However, we have 5 events in the environments 4 that violate the prohibition at different times and the 5th is an arbitrary event after the end of the contract. We do this to illustrate the range of time covered by the ”at any time repeated”. The result of the simulation is shown in 5.9.

```
result ContractState: noassets | ("Start Time" is 1,1,
    2018) ("End Time" is 31,12,2022) "NDA violation time"
    group is{("NDA violation time" # 4 is 31,12,2022) (
    "NDA violation time" # 3 is 30,12,2022) (
    "NDA violation time" # 2 is 26,5,2020)
    "NDA violation time" # 1 is 1,1,2018} |
    "Prohibition Clause" breached by 1,1,2023 with time
    recorded in "NDA violation time" | nopairs |
    notransactions
```

Figure 5.9: Case where the prohibition is breached

5.3 Validationthrough real world examples

In this chapter we specify real world examples of contracts in SlanC and evaluate them under different circumstances. The appeal of real world contracts is that they allow us to understand contracts better and assess the expressiveness of SlanC. We chose several contracts from different domains:

- A Parking Space Lease Agreement.
- A Bill Of Sale
- An Indemnity Agreement
- An Employment Agreement

5.3.1 Parking space lease agreement

A Parking Space Lease Agreement is between a landlord that controls an area of space, designated for a vehicle, and allows a person to rent it in return for payment. The contract can either be set for a fixed term or on a month to month

basis. The tenant shall only be able to store their vehicle on the premises unless otherwise agreed to by the landlord. There are many variations of this type of agreement, however, they all have the same general idea in common. We present one variation here, which has the following clauses:

- Items Left in Vehicle. Lessor shall not be responsible for damage or loss to possessions or items left in Lessee's vehicle.

- Damage to Vehicle. Lessor shall not be responsible for damage to Lessee's vehicle, whether or not such damage is caused by other vehicle(s) or person(s) in the parking lot and surrounding area.

- Parking Lot Attendants. Lessor shall not provide parking lot attendants. In the event that Lessor provides such attendants, any use of such attendant by Lessee to park or drive Lessee's vehicle shall be at Lessee's request, direction and sole risk of any resulting loss and Lessee shall indemnify Lessor for any loss resulting from such use.

- Payments by Lessee. Lessee agrees to pay $ Rent per month for the lease of the aforementioned parking space. Lessee is to make such leasehold payment –(to Lessor or Lessor's Agent) in person (or by mail) at Rent Mailing Address address. Payments shall be made in advance by Lessee on the first of each month.

- Receipts by Lessor. Lessor agrees to provide a receipt to Lessee for each payment received. Such receipt shall show the amount paid and number of

the leased parking space.

- Termination. Either party may terminate this Agreement by providing 30 days written notice to the other party. Any such notice shall be directed to a party at the party's address as listed below in this Agreement.

- Damages and Loss of Equipment. Lessee is responsible for any and all damages beyond normal wear and tear to the parking facilities. Lessee is also to be held responsible for replacement of any lost, stolen, damaged, or misplaced remote garage door openers or other parking facility related equipment lent to the Lessee by the Lessor.

Section 5.3.1.1 presents the specification of this contract in SlanC and section 5.3.1.2 presents its simulation and results using the semantics developed in Maude.

5.3.1.1 Specification in SlanC

Listing 5.9: Specification of the Parking Lease Agreement in SlanC

```
Contract
{
    Preamble
    {
      name: "Parking Lease agreement"
      start time: (1 , 1 , 2018)
      end time: (31 , 12 , 2018)
      parties: [
```

```
            { id: 'Lessee , name: "Some guy" , description: "
               ↪ someone who wants to rent a parking space" ,
               ↪ address: "some street - some building"}
            { id: 'Lessor , name: "Parking space owner" ,
               ↪ description: "someone who owns a parking space"
               ↪  , address: "The parking place avenue"}
          ]
  extends: none
}
Body
{
  ("1. Items Left in Vehicle" ; 'Lessor is not obligated to "
     ↪ replace damaged items left in lessee's car" for 'Lessee
     ↪  at any time repeated where time is "item damage time")
  ("2. Damage to Vehicle" ; 'Lessor is not obligated to "fix
     ↪ damage to Lessee's car" for 'Lessee at any time
     ↪ repeated where time is "car damage time")
  ("3. Parking Lot Attendants" ; 'Lessor is not obligated to "
     ↪ provide car attendants" for 'Lessee at any time
     ↪ repeated where time is "attendants time")
  ("4. Payments by Lessee" ; 'Lessee is obligated to transfer
     ↪ 100 USD for 'Lessor every (0 , 1 , 0) starting at (25 ,
```

```
    ↪  1 , 2018) and ending at (31 , 12 , 2018) where time is
    ↪  "Lessee payment time")
("5. Receipts by Lessor" ; 'Lessor is obligated to "send
    ↪ payment receipt" for 'Lessee at each "Lessee payment
    ↪ time" where time is "Receipt of payment time")
("6 Termination" ; any where time is "Termination time"
{
    ("Case where Lessor sends termination" ; if "sends
        ↪ termination notice" by 'Lessor for 'Lessee at any
        ↪ time is satisfied where time is "lessor termination
        ↪ notice" then all
    {
      "lessor termination clause" ; 'Lessor is permitted to "
          ↪ terminate agreement" for 'Lessee after ("lessor
          ↪ termination notice" + (0 , 1 , 0)) where time is "
          ↪ termination time"
    })
    ("Case where Lessee sends termination" ; if "sends
        ↪ termination notice" by 'Lessee for 'Lessor after any
        ↪  time is satisfied where time is "lessee termination
        ↪  notice" then all
    {
```

```
            ("lessee termination clause" ; 'Lessee is permitted to "
               ↪ terminate agreement" for 'Lessor after ("lessee
               ↪ termination notice" + (0 , 1 , 0)) where time is "
               ↪ termination time")
          })
      })
      ("7. Damages and Loss of Equipment." ; any where time is "
         ↪ damage and fix time" {
          ("7.1 Damage to parking facilities" ; 'Lessee is
             ↪ prohibited from "damage parking facilities" for '
             ↪ Lessor at any time repeated where time is "parking
             ↪  damage time")
          ("7.2 Fixing damages to parking facilities" ; 'Lessee is
             ↪  obligated to "fix damage to parking facilities"
             ↪ for 'Lessor at each "parking damage time" where
             ↪ time is "parking fix time")
        })
    }
}
```

In the specification of this contract, most of the clauses from the original contract map to a clause in the specification. Furthermore, most of clauses in the SlanC specification are trivial translations of the original ones, except for clauses

6 and 7. In 6, we state that if any of the clauses contained within the ”any” container is fulfilled then the container is fulfilled. These two clauses handle the two cases of initiating the termination of the contract, by the lessee and the lessor. Interestingly, both permissions contained within this clause declare the same time reference, which means that whoever terminates the contract first will define the time in this reference. In 7, the same container (”any”) is used to specify a prohibition and its reparation, by using the ”at any time repeated” time constraint in the prohibition and the ”at each [Time Reference]” construct in the obligation. This entire clause states that the lessee is not allowed to damage the facilities, but if they did then they must repair that damage immediately. The simulations give a more detailed description of the behavior of this clause.

5.3.1.2 Simulation in Maude

We simulated this contract in 3 different environments, as follows:

- All clauses are fulfilled / exercised.
- All clauses are breached / not exercised.
- All fulfilled except permissions.

All clauses are fulfilled / exercised An environment that reaches this state must contain at least one event for each of the clauses 1, 2, 3, 5 and 6 and all the events that satisfy clause 4 and no events for clause 7, as follows:

```
( ("replace damaged items left in lessee's car" by 'Lessor for '
   ↪ Lessee) @ (1 , 1 , 2018) )
```

```
( ("replace damaged items left in lessee's car" by 'Lessor for '
   ↪ Lessee) @ (20 , 2 , 2018) )
( ( "fix damage to Lesse's car" by 'Lessor for 'Lessee) @ (24 , 1 ,
   ↪  2018) )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 1 , 2018) in
   ↪  fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 1 , 2018)
   ↪  )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 2 , 2018) in
   ↪  fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 2 , 2018)
   ↪  )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 3 , 2018) in
   ↪  fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 3 , 2018)
   ↪  )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 4 , 2018) in
   ↪  fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 4 , 2018)
   ↪  )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 5 , 2018) in
   ↪  fulfillment of "4. Payments by Lessee" )
```

```
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 5 , 2018)
   ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 6 , 2018) in
   ↪ fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 6 , 2018)
   ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 7 , 2018) in
   ↪ fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 7 , 2018)
   ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 8 , 2018) in
   ↪ fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 8 , 2018)
   ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 9 , 2018) in
   ↪ fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 9 , 2018)
   ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 10 , 2018)
   ↪ in fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 10 ,
   ↪ 2018) )
```

```
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 11 , 2018)
   ↪ in fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 11 ,
   ↪ 2018) )
( ("sends termination notice" by 'Lessor for 'Lessee) @ (29 , 11 ,
   ↪ 2018) )
( ("sends termination notice" by 'Lessee for 'Lessor) @ (30 , 11 ,
   ↪ 2018) )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 12 , 2018)
   ↪ in fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 12 ,
   ↪ 2018) )
( ("provide car attendants" by 'Lessor for 'Lessee) @ (25 , 12 ,
   ↪ 2018) )
( ("terminate agreement" by 'Lessor for 'Lessee) @ (29 , 12 , 2018)
   ↪ )
( ("terminate agreement" by 'Lessee for 'Lessor) @ (30 , 12 , 2018)
   ↪ )
( ("xcv" by 'Lessee for 'Lessor) @ (1 , 1 , 2019))
```

Figure 5.10 shows the result of this simulation. In the resulting state, there are no assets because no assets list was given as input. The variable list shows the time references and their values that were defined by the given environment.

The final status of each clause is also shown, where each one is marked fulfilled. The clause asset pair list shows that there is only one clause that involves asset transfer, and nothing remains to be transferred for that clause to be fulfilled. Finally, the list of all transactions, which represent all the lease payments.

This represents the happy path for this contract where all the clauses are fulfilled/exercised and there are no semantic problem. There reason there might be a semantic problem is the presence of a termination clause. This termination clause could present a semantic problem if it was fulfilled at an earlier time, because SlanC does not understand that it terminates the contract. This call for implementing a new special actions that represent contract termination, which the current version of SlanC lacks.

Figure 5.10: All clauses are fulfilled / exercised

All clauses are breached / not exercised An environment that results in this state is the opposite of the one presented in the previous case. Which contains

no events for clauses 1, 2, 3, 5 and 6, and at least one missing event for clause 4 and at least one event for clause 7, as follows:

```
( "damage parking facilities" by 'Lessee for 'Lessor @ (31 , 12 ,
    ↪ 2018))
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 1 , 2018) in
    ↪  fulfillment of "4. Payments by Lessee" )
( ("Arbitrary Event!!" by 'Lessor for 'Lessee) @ (1 , 1 , 2019) )
```

This results in a breach for each of the agreements, because the lease transfers are not complete and damage was done to the parking facilities with no fixes and no receipts were delivered for the payments. Furthermore, an interesting observation about this simulation is that the termination clause is not fulfilled, therefore semantically this contract must renew for another term automatically after its end time, however it does not. This in part due to the limitation we mentioned in the previous case and also because SlanC does not support having multiple states for a single contract, which is necessary to fully support auto contract renewal.

All fulfilled except permissions In this case we include all the events that satisfy the obligations while omitting all the events that satisfy permissions and break prohibitions, as follows:

```
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 1 , 2018) in
    ↪  fulfillment of "4. Payments by Lessee" )
```

```
result ContractState: noassets | ("Start Time" is 1,1,2018) ("End Time" is 31,12,2018) (
    "parking damage time" group is{"parking damage time" # 1 is 31,12,2018})
    "Lessee payment time" group is{"Lessee payment time" # 1 is 25,1,2018} | (
    "7. Damages and Loss of Equipment." breached by 1,1,2019 with time recorded in
    "damage and fix time") ("7.2 Fixing damages to parking facilities" breached by 1,1,2019
    with time recorded in "parking fix time") ("7.1 Damage to parking facilities" breached by
    1,1,2019 with time recorded in "parking damage time") (
    "Case where Lessee sends termination-CONSEQUENT" breached by 1,1,2019 with time recorded
    in "lessee termination notice-CONSEQUENT") (
    "Case where Lessee sends termination-CONDITION" breached by 1,1,2019 with time recorded in
    "lessee termination notice") ("Case where Lessor sends termination-CONSEQUENT" breached by
    1,1,2019 with time recorded in "lessor termination notice-CONSEQUENT") (
    "Case where Lessor sends termination-CONDITION" breached by 1,1,2019 with time recorded in
    "lessor termination notice") ("5. Receipts by Lessor" breached by 1,1,2019 with time
    recorded in "Receipt of payment time") ("4. Payments by Lessee" breached by 1,1,2019 with
    time recorded in "Lessee payment time") ("3. Parking Lot Attendants" not exercised by 1,1,
    2019 with time recorded in "attendants time") ("2. Damage to Vehicle" not exercised by 1,
    1,2019 with time recorded in "car damage time") "1. Items Left in Vehicle" not exercised
    by 1,1,2019 with time recorded in "item damage time" | "4. Payments by Lessee" ; 1100 |
    transaction ; (transfer 100 USD by 'Lessee for 'Lessor @ 25,1,2018 in fulfillment of
    "4. Payments by Lessee")
```

Figure 5.11: All clauses are breached / not exercised

```
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 1 , 2018)
    ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 2 , 2018) in
    ↪ fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 2 , 2018)
    ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 3 , 2018) in
    ↪ fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 3 , 2018)
    ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 4 , 2018) in
    ↪ fulfillment of "4. Payments by Lessee" )
```

```
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 4 , 2018)
   ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 5 , 2018) in
   ↪ fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 5 , 2018)
   ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 6 , 2018) in
   ↪ fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 6 , 2018)
   ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 7 , 2018) in
   ↪ fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 7 , 2018)
   ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 8 , 2018) in
   ↪ fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 8 , 2018)
   ↪ )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 9 , 2018) in
   ↪ fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 9 , 2018)
   ↪ )
```

```
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 10 , 2018)
   ↪ in fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 10 ,
   ↪ 2018) )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 11 , 2018)
   ↪ in fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 11 ,
   ↪ 2018) )
( ("sends termination notice" by 'Lessor for 'Lessee) @ (29 , 11 ,
   ↪ 2018) )
( ("sends termination notice" by 'Lessee for 'Lessor) @ (30 , 11 ,
   ↪ 2018) )
( ( (transfer 100 USD) by 'Lessee for 'Lessor) @ (25 , 12 , 2018)
   ↪ in fulfillment of "4. Payments by Lessee" )
( ("send payment receipt" by 'Lessor for 'Lessee) @ (25 , 12 ,
   ↪ 2018) )
( ("xcv" by 'Lessee for 'Lessor) @ (1 , 1 , 2019))
```

Figure 5.12 shows the results of the simulation, where every clause is fulfilled except the permissions. However, since the permissions in clause 7 are not fulfilled which results in the consequent of their respective conditions being unfulfilled. Beyond that, there are no differences between this case and the first one.

Figure 5.12: All fulfilled except permissions

5.3.2 Indemnity agreement

An indemnity agreement (sometimes called a ”hold harmless agreement”) can be a contract or a section of a contract. In these cases, an indemnity agreement is contract language that indemnifies (holds harmless) one of the parties in a contract for specific actions that might cause damage to the other party. For example, a landlord may require a tenant to sign a ”hold harmless” clause in a rental agreement, agreeing that the landlord is not responsible for damages caused by the tenant's negligence. In this case, the landlord must be persuaded to sign a lease contract which could cause him or her to be sued. Therefore, the tenant is being required to indemnify the landlord, so the contract can be signed.

There many variations of this type of agreement depending on the type of contract it exists in, e.g construction, rental etc... However, they all have this general idea in common: Party **A** could potentially be sued by Party **B** for damages, even

though **A** might have had nothing to do with the damage **B** received, therefore **B** must indemnify **A**.

We choose a generic indemnity agreement that has the following clauses:

- Agreement of Indemnification dated **1/1/2018** between **John Doe(Undersigned)** and **Jane Smith (Indemnitees)**.
- For value received, **John Doe (Undersigned)** jointly and severally agree to indemnify and save harmless Agreement of Indemnification dated 1/1/2018 between **John Doe(Undersigned)** and **Jane Smith (Indemnitees)** and its successors and assigns from any claim, action, liability, loss, damage or suit arising from negligence or natural disasters.
- Thereafter, **John Doe (Undersigned)** shall at its own expense defend, protect and save harmless **Jane Smith (Indemnitees)** against said claim or any loss or liability resulting therefrom.
- Should **John Doe (Undersigned)** fail to so defend and / or indemnify and save harmless, then, in such case, **Jane Smith (Indemnitees)** shall have full rights to defend, pay or settle said claim on their own behalf without notice to **John Doe (Undersigned)** for all fees, costs, and payments made or agreed to be paid to discharge said claim.
- **John Doe (Undersigned)** agrees to pay all reasonable attorneys' fees necessary to enforce said indemnification.
- This agreement shall be unlimited as to amount or duration, and it shall

be binding upon and inure to the benefit of the parties, their successors, assigns and personal agents and representatives.

Section 5.3.2.1 presents the specification of this contract in SlanC and section 5.3.2.2 presents its simulation and results using the semantics developed in Maude.

5.3.2.1 Specification in SlanC

This contract can be specified in SlanC as shown in Figure (). All of the clauses of the indemnity agreement could be specified except for the clause stating that the agreement is "unlimited by time", which can't be expressed in SlanC, although we could change the end time of the contract with an arbitrarily large date (e.g 1/1/9999). Furthermore, there is a difference in style between the original and the SlanC specification, where the specification is much less verbose than the original, which can a detriment when specifying contracts, because looking at the specification alone it is not clear what is meant by "harm" or "legitimate claim".

The specification actually states that the undersigned can make more than one "legitimate claim" and they are prohibited from any and all "harm". Also, they have to repair each "harm" they cause and they have to pay all the fees to make the indemnification possible.

Listing 5.10: Specification of an Indemnity Agreement in SlanC

```
Contract
{
   Preamble
```

```
{
  name: "Indemnity Agreement"
  start time: (1 , 1 , 2018)
  end time: (31 , 12 , 2018)
  parties: [
           { id: 'Undersigned , name: "Some Undersigned" ,
              ↪ description: "someone who is an Undersigned" ,
              ↪ address: "some street - some building"}
           { id: 'Indemnitee , name: "an Indemnitee" ,
              ↪ description: "someone who is an Indemnitee" ,
              ↪ address: "Indemnitee avenue"}
         ]
  extends: none
}
Body
{
  ("1. Legitimate claims" ; 'Undersigned is permitted to "make
     ↪ legitimate claim" for 'Indemnitee at any time repeated
     ↪ where time is "legitimate claim time")
  ("2. Prohibition from harm" ; 'Undersigned is prohibited from
     ↪  "harm" for 'Indemnitee at any time repeated where time
     ↪  is "harm time")
```

```
      ("3. Repairing harm" ; 'Undersigned is obligated to "fix harm
         ↪ " for 'Indemnitee at each "harm time" where time is "
         ↪ fix harm time")
      ("4. Attorneys fees" ; 'Undersigned is obligated to "pay
         ↪ attorneys fess to make indemnification possible" for '
         ↪ Indemnitee at "Start Time" where time is "pay
         ↪ indemnification fee time")
  }
}
```

5.3.2.2 Simulation in Maude

We simulated this contract in different environments, as follows:

- Fulfillment case where no legitimate claims are made.
- Fulfillment case where legitimate claims are made and damage is done and fixed.
- Breach case where no attorneys fees are paid.
- Breach case where harm is done but no fix is done.

It is important to note that SlanC does not not if each of these is a fulfillment or breach case, because SlanC deals at the clause level and it does not make any assertions about the contract as a whole.

Fulfillment case where no legitimate claims are made. There are multiple fulfillment states that can be reached given this condition. One such case is when no legitimate claims are made and no harm is done and all attorneys fees are paid. The environment that fulfills this state is:

```
( ("pay attorneys fess to make indemnification possible" by '
   ↪ Undersigned for 'Indemnitee) @ (1 , 1 , 2018))
( ("xcv" by 'Undersigned for 'Indemnitee) @ (1 , 1 , 2019))
```

Figure 5.13 shows the results of this simulation, where all the agreements are fulfilled except the legitimate claims permission. Interestingly, the third clause which is about repairing harm is fulfilled although no harm was done. This reason for this is the ”at each [Time Reference]” time constraint which is especially implemented for these types of cases. In other words, the Undersigned must fix every harm they have done, but if they have done no harm they must not be compelled to fix anything.

```
result ContractState: noassets | ("pay indemnification fee time"
    is 1,1,2018) ("Start Time" is 1,1,2018) "End Time" is 31,12,
    2018 | ("3. Repairing harm" fulfilled by 1,1,2019 with time
    recorded in "fix harm time") ("2. Prohibition from harm"
    fulfilled by 1,1,2019 with time recorded in "harm time") (
    "1. Legitimate claims" not exercised by 1,1,2019 with time
    recorded in "legitimate claim time") "4. Attorneys fees"
    fulfilled by 1,1,2018 with time recorded in
    "pay indemnification fee time" | nopairs | notransactions
```

Figure 5.13: Fulfillment case where no legitimate claims are made.

Fulfillment case where legitimate claims are made and damage is done and fixed This case is exactly like the previous case, except that there was some damage done by the Undersigned but it was fixed. One such environment that produces this state is shown below:

```
( ("pay attorneys fess to make indemnification possible" by '
   ↪ Undersigned for 'Indemnitee) @ (1 , 1 , 2018))
( ("harm" by 'Undersigned for 'Indemnitee) @ (9 , 3 , 2018))
( ("fix harm" by 'Undersigned for 'Indemnitee) @ (9 , 3 , 2018))
( ("harm" by 'Undersigned for 'Indemnitee) @ (20 , 5 , 2018))
( ("fix harm" by 'Undersigned for 'Indemnitee) @ (20 , 5 , 2018))
( ("xcv" by 'Undersigned for 'Indemnitee) @ (1 , 1 , 2019))
```

Figre 5.14 shows the result of this simulation, where the only difference is that the prohibition from doing harm is breached multiple times.

```
result ContractState: noassets | ("pay indemnification fee time" is 1,1,2018) (
    "Start Time" is 1,1,2018) ("End Time" is 31,12,2018) ("harm time" group is{(
    "harm time" # 2 is 20,5,2018) "harm time" # 1 is 9,3,2018}) "fix harm time"
    group is{("fix harm time" # 2 is 20,5,2018) "fix harm time" # 1 is 9,3,2018} |
    ("3. Repairing harm" fulfilled by 1,1,2019 with time recorded in
    "fix harm time") ("2. Prohibition from harm" breached by 1,1,2019 with time
    recorded in "harm time") ("1. Legitimate claims" not exercised by 1,1,2019
    with time recorded in "legitimate claim time") "4. Attorneys fees" fulfilled
    by 1,1,2018 with time recorded in "pay indemnification fee time" | nopairs |
    notransactions
```

Figure 5.14: Fulfillment case where legitimate claims are made and damage is done and fixed

Breach case where no attorneys fees are paid. In this case all the agreements are breached except for the obligation to fix damages, which is fulfilled because no harm is done. The environment, which consists of a single arbitrary

event, that results in this state is shown and Figure 5.15 shows the result of this simulation.

```
( ("Arbitrary Event!" by 'Undersigned for 'Indemnitee) @ (1 , 1 ,
    ↪ 2019))
```

```
result ContractState: noassets | ("Start Time" is 1,1,2018) "End Time"
    is 31,12,2018 | ("4. Attorneys fees" breached by 1,1,2019 with time
    recorded in "pay indemnification fee time") ("3. Repairing harm"
    fulfilled by 1,1,2019 with time recorded in "fix harm time") (
    "2. Prohibition from harm" fulfilled by 1,1,2019 with time recorded
    in "harm time") "1. Legitimate claims" not exercised by 1,1,2019
    with time recorded in "legitimate claim time" | nopairs |
    notransactions
```

Figure 5.15: Breach case where no attorneys fees are paid.

Breach case where harm is done but no fix is done. This case is exactly like the previous case with one major difference, the prohibition from doing harm and the obligation to fix that harm is breached, as follows:

```
( ("pay attorneys fess to make indemnification possible" by '
    ↪ Undersigned for 'Indemnitee) @ (1 , 1 , 2018))
( ("harm" by 'Undersigned for 'Indemnitee) @ (9 , 3 , 2018))
( ("fix harm" by 'Undersigned for 'Indemnitee) @ (9 , 3 , 2018))
( ("harm" by 'Undersigned for 'Indemnitee) @ (20 , 5 , 2018))
( ("Arbitrary Event!" by 'Undersigned for 'Indemnitee) @ (1 , 1 ,
    ↪ 2019))
```

Figure 5.16 shows the results of the simulation, where two harm events were recorded in the variable list and a single fix event is recorded. This mismatch

coupled with the knowledge that the contract has ended leads to the conclusion that the obligation of fixng harm is breached.

```
result ContractState: noassets | ("pay indemnification fee time" is 1,1,2018)
    ("Start Time" is 1,1,2018) ("End Time" is 31,12,2018) ("harm time" group
    is{("harm time" # 2 is 20,5,2018) "harm time" # 1 is 9,3,2018})
    "fix harm time" group is{"fix harm time" # 1 is 9,3,2018} | (
    "3. Repairing harm" breached by 1,1,2019 with time recorded in
    "fix harm time") ("2. Prohibition from harm" breached by 1,1,2019 with
    time recorded in "harm time") ("1. Legitimate claims" not exercised by 1,
    1,2019 with time recorded in "legitimate claim time") "4. Attorneys fees"
    fulfilled by 1,1,2018 with time recorded in
    "pay indemnification fee time" | nopairs | notransactions
```

Figure 5.16: Breach case where harm is done but no fix is done.

5.3.3 Bill Of Sale

A Bill of Sale is a form that a seller uses to document the sale of an item to a buyer. It serves as a receipt for personal sales and purchases and includes buyer and seller information and details about the goods, its location, and the price. A Bill of Sale should only be used for ”as-is” purchases, when payment in full will is made once the item is exchanged. It expressly disclaims any warranties that relate to the quality or fitness of the product. The bill of sale that we specify has the following clauses:

1. IN CONSIDERATION of Jane Smith of Jane Smith Street (the ’Purchaser’) providing $50,000.00 USD, which includes all sales taxes (the ’Purchase Price’), the receipt and sufficiency of which is hereby acknowledged to John Doe of John Doe Street. (the ’Seller’), the Seller SELLS AND DELIVERS the Property to the Purchaser.

2. PURCHASE PRICE: The Purchaser will pay the Purchase Price to the Seller by cash.

3. PROPERTY: The Seller will sell and deliver to the Purchaser the following personal property (the 'Property'): A house.

4. WARRANTIES: The Seller warrants that the Property is free of any liens and encumbrances and that the Seller is the legal owner of the Property. The Seller also warrants that the Seller has the full right and authority to sell and deliver the Property and that the Seller will defend the title of the Property against any and all claims and demands.

5. 'AS IS' CONDITION: The Purchaser acknowledges that the Property is sold 'as is'. The Seller expressly disclaims any implied warranty as to fitness for a particular purpose and any implied warranty as to merchantability. The Seller expressly disclaims any expressed or other implied warranties.

6. WORKING ORDER: Any warranty as to the condition of the Property is expressly disclaimed by the Seller.

7. MANUFACTURER'S WARRANTY: Any disclaimer of warranties by the Seller in this Bill of Sale will not in any way affect the terms of any applicable warranties from the manufacturer of the Property.

8. LIABILITIES: The Seller does not assume, nor does the Seller authorize any other person on the behalf of the Seller to assume, any liability in connection with the sale or delivery of the Property.

9. INSPECTION: The Purchaser accepts the Property in its existing condition given that the Purchaser has either inspected the Property or was given the opportunity to inspect the Property but chose to not inspect it.

Section 5.3.3.1 presents the specification of this contract in SlanC and section 5.3.3.2 presents its simulation and results using the semantics developed in Maude.

5.3.3.1 Specification in SlanC

The specification for a bill of sale is shown in Listing 5.11. In this contract, most of the agreements form the original map to single agreements in the specification except for the last one which maps into two agreements. The reason for this is that the original agreement state that the bill of sale is governed by some law, which implies that both parties agree to this fact. Therefore, two obligations are needed to specify this fact with the same action but different directions, as shown in the specification clauses 9 and 10. Furthermore, contrary to the previous example, this specification is verbose which helps clarify the meaning behind some of the clauses, however, the events that trigger those verbose clauses have to match their verbosity, as we will show in the simulations that we performed. Moreover, an interesting observation about this bill of sale is that it does not have an end time attached to it, yet in the SlanC specification we chose an arbitrary end time. The end time in this case does not have any meaning because all the agreements within the specification do not use it, but it has to be included as part of the preamble.

Listing 5.11: Specification of a Bill of Sale in SlanC

```
Contract
{
    Preamble
    {
      name: "Bill of Sale"
      start time: (1 , 1 , 2018)
      end time: (2 , 1 , 2018)
      parties: [
                { id: 'Purchaser , name: "Some Purchaser" ,
                    ↪ description: "Some Purchaser" , address: "some
                    ↪ street - some building"}
                { id: 'Seller , name: "Some Seller" , description: "
                    ↪ Some Seller" , address: "Some Seller avenue"}
               ]
      extends: none
    }
    Body
    {
      ("1. Money Transfer" ; 'Purchaser is obligated to transfer
          ↪ 50000 USD for 'Seller at "Start Time" where time is "
          ↪ Money Transfer Time")
```

```
("2. Property Transfer" ; 'Seller is obligated to "transfer
   ↪ the property" for 'Purchaser at "Money Transfer Time"
   ↪ where time is "Property Transfer Time")
("3. Warranty" ; 'Seller is obligated to "give warranty that
   ↪ the property is free of any liens and encumbrances and
   ↪ that the Seller is the legal owner of the Property. The
   ↪  Seller also warrants that the Seller has the full
   ↪ right and authority to sell and deliver the Property
   ↪ and that the Seller will defend the title of the
   ↪ Property against any and all claims and demands." for '
   ↪ Purchaser at "Property Transfer Time" where time is "
   ↪ Warranty Time")
("4. As is Condition" ; 'Purchaser is obligated to "
   ↪ acknowledge that the Property is sold 'as is'" for '
   ↪ Seller at "Property Transfer Time" where time is "As is
   ↪  acknowledgement Time")
("5. Working Order" ; 'Purchaser is obligated to "acknowledge
   ↪  any warranty as to the condition of the property is
   ↪ expressly disclaimed by the seller" for 'Seller at "
   ↪ Property Transfer Time" where time is "Working order
   ↪ acknowledgement Time")
```

```
("6. Manufacturers Warranty" ; 'Seller is obligated to "
   ↪ acknowledge Any disclaimer of warranties by the Seller
   ↪ in this Bill of Sale will not in any way affect the
   ↪ terms of any applicable warranties from the
   ↪ manufacturer of the Property" for 'Purchaser at "
   ↪ Property Transfer Time" where time is " Manufacturers
   ↪ Warranty acknowledgement Time")
("7. Liabilities" ; 'Seller is not obligated to "assume any
   ↪ liability in connection with the sale or delivery of
   ↪ the Property" for 'Purchaser at "Property Transfer Time
   ↪ " where time is "Liability Time")
("8. Inspection" ; 'Purchaser is obligated to "accept the
   ↪ Property in its existing condition given that the
   ↪ Purchaser has either inspected the Property or was
   ↪ given the opportunity to inspect the Property but chose
   ↪  to not inspect it" for 'Seller at "Property Transfer
   ↪ Time" where time is "Inspection Time")
("9. Governing Law Acknowledgement by Seller" ; 'Seller is
   ↪ obligated to "acknowledge that the bill of sale is
   ↪ governed by the laws of the State of Alabama" for '
   ↪ Purchaser at "Property Transfer Time" where time is "
   ↪ Governing Law Acknowledgement by Seller Time")
```

```
      ("10. Governing Law Acknowledgement by Purchaser" ; '
         ↪ Purchaser is obligated to "acknowledge that the bill of
         ↪  sale is governed by the laws of the State of Alabama"
         ↪ for 'Seller at "Property Transfer Time" where time is "
         ↪ Governing Law Acknowledgement by Purchaser Time")
   }
}
```

5.3.3.2 Simulation in Maude

We simulate this contract in 2 different cases:

- Fulfillment case where all clauses are fulfilled.
- Breach case where all clauses are breached.

Case where all clauses are fulfilled. This is another straightforward case where we simulate the contract in an environment that satisfies all its clauses. An environment that satisfies this is shown below.

```
( (transfer 50000 USD by 'Purchaser for 'Seller) @ (1 , 1 , 2018)
   ↪ in fulfillment of "1. Money Transfer")
( ("transfer the property" by 'Seller for 'Purchaser) @ (1 , 1 ,
   ↪ 2018))
( ("give warranty that the property is free of any liens and
   ↪ encumbrances and that the Seller is the legal owner of the
```

```
    ↪ Property. The Seller also warrants that the Seller has the
    ↪ full right and authority to sell and deliver the Property
    ↪ and that the Seller will defend the title of the Property
    ↪ against any and all claims and demands." by 'Seller for '
    ↪ Purchaser) @ (1 , 1 , 2018))
( ("acknowledge that the Property is sold 'as is'" by 'Purchaser
    ↪ for 'Seller) @ (1 , 1 , 2018))
( ("acknowledge any warranty as to the condition of the property is
    ↪  expressly disclaimed by the seller" by 'Purchaser for '
    ↪ Seller) @ (1 , 1 , 2018))
( ("acknowledge Any disclaimer of warranties by the Seller in this
    ↪ Bill of Sale will not in any way affect the terms of any
    ↪ applicable warranties from the manufacturer of the Property"
    ↪  by 'Seller for 'Purchaser) @ (1 , 1 , 2018))
( ("assume any liability in connection with the sale or delivery of
    ↪  the Property" by 'Seller for 'Purchaser) @ (1 , 1 , 2018))
( ("accept the Property in its existing condition given that the
    ↪ Purchaser has either inspected the Property or was given the
    ↪  opportunity to inspect the Property but chose to not
    ↪ inspect it" by 'Purchaser for 'Seller) @ (1 , 1 , 2018))
( ("acknowledge that the bill of sale is governed by the laws of
    ↪ the State of Alabama" by 'Purchaser for 'Seller) @ (1 , 1 ,
```

```
    ↪ 2018))
( ("acknowledge that the bill of sale is governed by the laws of
    ↪ the State of Alabama" by 'Seller for 'Purchaser) @ (1 , 1 ,
    ↪ 2018))
( ("xcv" by 'Seller for 'Purchaser) @ (1 , 1 , 2019))
```

Figure 5.17 shows the fulfillment state for this contract.

```
result ContractState: noassets | ("Governing Law Acknowledgement by Seller Time" is 1,1,2018)
    ("Governing Law Acknowledgement by Purchaser Time" is 1,1,2018) ("Inspection Time" is 1,1,
    2018) ("Liability Time" is 1,1,2018) (" Manufacturers Warranty acknowledgement Time" is 1,
    1,2018) ("Working order acknowledgement Time" is 1,1,2018) ("As is acknowledgement Time"
    is 1,1,2018) ("Warranty Time" is 1,1,2018) ("Property Transfer Time" is 1,1,2018) (
    "Money Transfer Time" is 1,1,2018) ("Start Time" is 1,1,2018) "End Time" is 31,12,2018 | (
    "9. Governing Law Acknowledgement by Seller" fulfilled by 1,1,2018 with time recorded in
    "Governing Law Acknowledgement by Seller Time") (
    "10. Governing Law Acknowledgement by Purchaser" fulfilled by 1,1,2018 with time recorded
    in "Governing Law Acknowledgement by Purchaser Time") ("8. Inspection" fulfilled by 1,1,
    2018 with time recorded in "Inspection Time") ("7. Liabilities" exercised by 1,1,2018 with
    time recorded in "Liability Time") ("6. Manufacturers Warranty" fulfilled by 1,1,2018 with
    time recorded in " Manufacturers Warranty acknowledgement Time") ("5. Working Order"
    fulfilled by 1,1,2018 with time recorded in "Working order acknowledgement Time") (
    "4. As is Condition" fulfilled by 1,1,2018 with time recorded in
    "As is acknowledgement Time") ("3. Warranty" fulfilled by 1,1,2018 with time recorded in
    "Warranty Time") ("2. Property Transfer" fulfilled by 1,1,2018 with time recorded in
    "Property Transfer Time") "1. Money Transfer" fulfilled by 1,1,2018 with time recorded in
    "Money Transfer Time" | "1. Money Transfer" ; 0 | transaction ; (transfer 50000 USD by
    'Purchaser for 'Seller @ 1,1,2018 in fulfillment of "1. Money Transfer")
```

Figure 5.17: Case where all clauses are fulfilled.

Case where all clauses are breached. The breach state for this contract is quite interesting because it is not the one resulting from an environment that is the exact opposite of the one in the fulfillment case. In fact, trying the opposite environment (Listing 5.12) will yield the state shown in Figure 5.18. The result shows that only one agreement is breached with nothing about the other agreements as though they were not in the contract. This is because the second agreement which is about the transfer of the property uses ”Money Transfer Time” in its

time constraint, which is declared by the ”1. Money Transfer” agreement, which is breached meaning that there is no fulfillment time for it in the variable list. This means that the second agreement (”2. Property Transfer”) states that the property transfer must happen at an undefined time, which is impossible, therefore, this can never be fulfilled or breached. Furthermore, all the other agreements use ”Property Transfer Time” which is declared by the second agreement, which is undefined, hence the rest, also, can never be fulfilled or breached.

Listing 5.12: Environment with a single arbitrary event after the end of the contract

```
( ("xcv" by 'Seller for 'Purchaser) @ (1 , 1 , 2019))
```

```
result ContractState: noassets | ("Start Time" is 1,1,2018)
    "End Time" is 31,12,2018 | "1. Money Transfer" breached by 1,
    1,2019 with time recorded in "Money Transfer Time" | nopairs
    | notransactions
```

Figure 5.18: Case where only the first is breached.

Given that if the first agreement is not fulfilled then no assertion can be made about the other agreements, hence, we introduce the event that makes the first agreement fulfilled as shown in Listing 5.13. The resulting state (Figure 5.19) shows that the first agreement is fulfilled and the second is breached and nothing about the rest. This is expected because the other agreements depend on the second agreement.

Listing 5.13: Environment with event fulfilling the first agreement

```
( (transfer 50000 USD by 'Purchaser for 'Seller) @ (1 , 1 , 2018)
```

```
    ↪ in fulfillment of "1. Money Transfer")
( ("transfer the property" by 'Seller for 'Purchaser) @ (2 , 1 ,
    ↪ 2018))
( ("xcv" by 'Seller for 'Purchaser) @ (1 , 1 , 2019))
```

```
result ContractState: noassets | ("Money Transfer Time" is 1,1,2018) (
    "Start Time" is 1,1,2018) "End Time" is 31,12,2018 | (
    "2. Property Transfer" breached by 2,1,2018 with time recorded in
    "Property Transfer Time") "1. Money Transfer" fulfilled by 1,1,2018
    with time recorded in "Money Transfer Time" | "1. Money Transfer" ; 0
    | transaction ; (transfer 50000 USD by 'Purchaser for 'Seller @ 1,1,
    2018 in fulfillment of "1. Money Transfer")
```

Figure 5.19: Case where the first agreement is fulfilled and the second is breached.

Given the information from the previous two simulations, we add another event that fulfills the second agreement to the environment (Listing 5.14) and run the simulation, to obtain the state shown in Figure 5.20. The resulting state shows that the first two agreements are fulfilled while the rest are breached, as we expect because the dependency of agreements 3 through 10 is fulfilled.

Listing 5.14: Environment with event fulfilling the first agreement

```
( (transfer 50000 USD by 'Purchaser for 'Seller) @ (1 , 1 , 2018)
    ↪ in fulfillment of "1. Money Transfer")
( ("transfer the property" by 'Seller for 'Purchaser) @ (1 , 1 ,
    ↪ 2018))
( ("xcv" by 'Seller for 'Purchaser) @ (1 , 1 , 2019))
```

```
result ContractState: noassets | ("Property Transfer Time" is 1,1,2018) (
    "Money Transfer Time" is 1,1,2018) ("Start Time" is 1,1,2018) "End Time" is 31,12,2018
    | ("10. Governing Law Acknowledgement by Purchaser" breached by 3,1,2018 with time
    recorded in "Governing Law Acknowledgement by Purchaser Time") (
    "9. Governing Law Acknowledgement by Seller" breached by 3,1,2018 with time recorded in
    "Governing Law Acknowledgement by Seller Time") ("8. Inspection" breached by 3,1,2018
    with time recorded in "Inspection Time") ("7. Liabilities" not exercised by 3,1,2018
    with time recorded in "Liability Time") ("6. Manufacturers Warranty" breached by 3,1,
    2018 with time recorded in " Manufacturers Warranty acknowledgement Time") (
    "5. Working Order" breached by 3,1,2018 with time recorded in
    "Working order acknowledgement Time") ("4. As is Condition" breached by 3,1,2018 with
    time recorded in "As is acknowledgement Time") ("3. Warranty" breached by 3,1,2018 with
    time recorded in "Warranty Time") ("2. Property Transfer" fulfilled by 1,1,2018 with
    time recorded in "Property Transfer Time") "1. Money Transfer" fulfilled by 1,1,2018
    with time recorded in "Money Transfer Time" | "1. Money Transfer" ; 0 | transaction ; (
    transfer 50000 USD by 'Purchaser for 'Seller @ 1,1,2018 in fulfillment of
    "1. Money Transfer")
```

Figure 5.20: Case where all clauses are breached except the first two.

5.3.4 Employment Agreement

The purpose of an employment agreement is to set out the terms and conditions of the relationship between an employer and an employee. It states the obligations they have to each other and the benefits they will receive from each other. Employment agreements do not need to repeat terms and conditions set out in the organization's policies. Employment agreements may need to be more detailed if policies are not in place or if the particular position has specific requirements. The employment agreement that we specify here is shown in Figure **??**.

Section 5.3.4.1 presents the specification of this agreement in SlanC and section 5.3.4.2 presents its simulation and results using the semantics developed in Maude.

5.3.4.1 Specification in SlanC

There are many differences between the original employment agreement and its specification in SlanC, because each of its clauses implicitly state that must be stated explicitly when using SlanC. The following discusses each clause from the

original agreement and why it was translated to SlanC in that way.

- ”1. Employment” and ”2. Position”: This clause translates to an ”all” container in ”1. Employment and Position”, because they encapsulates different sub-clauses that must all be fulfilled. Each of these sub-clauses is given as a simple agreement, as shown in the specification. Furthermore, the clauses 1 and 2 were combined because the latter is a mere elaboration on the portion of the former that discusses the position.

- ”3. Compensation”: This clause to an ”all” container and two additional clauses in 2, 2.98 and 2.99 as shown in the specification. Each of these agreements is a straight forward translation from the original, except of the the last agreement 2.99 which is implicitly understood from the original agreement.

- ”4, 5, 6 and 7”: These also map to single agreements in the specification to clauses 3, 4, 5 and 6, respectively.

- ”8. Termination”: Termination translates to an ”any” container and a separate agreement in the specification to clauses 7 and 7.99 . The clauses within 7 in the specification are describe the different scenarios of termination. The last clause (7.99) states that the employee has to return employers property upon termination. This clause (”7.”) and ”5. Probation” present an interesting way of using time references, because they declare the same time reference. This way if either one of those is fulfilled it will trigger clause ”7.99”.

- ”9. Non Competition”: This clause is also translated into an ”all” container in the specification, which contains straightforward translation of the sub-clauses into SlanC.

- 10, 11, 12 and 13: Each of these clauses translate to an ”all” container that includes an obligation by each party to do the same action. This is done be cause the original clauses state that both parties need to agree on something or acknowledge something, such as governing laws.

Another interesting aspect of this contract is that it extends the non disclosure agreement we specified and simulated in sections 5.2.3.1 and 5.2.3.2, respectively.

Listing 5.15: Specification of an Employment Agreement in SlanC

```
Contract
{
    Preamble
    {
      name: "Employment Agreement"
      start time: (1 , 1 , 2018)
      end time: (31 , 12 , 2018)
      parties: [
               { id: 'Employer , name: "Some Employer" , description
                  ↪ : "someone who is Employer" , address: "some
                  ↪ street - some building"}
               { id: 'Employee , name: "an Employee" , description:
```

```
                ↪ "someone who is an Employee" , address: "
                ↪ Employee avenue"}
            ]
  extends: "Non disclosure agreement"
}
Body
{
  ("1. Employment" ; all where time is "Employment Clause
     ↪ Fulfillment Time"
    {
      ("1.0 Position" ; 'Employer is obligated to "provide a
         ↪ Senior Software Engineer Position" for 'Employee at
         ↪ "Start Time" where time is "Position Time")
      ("1.1 Duties" ; 'Employee is obligated to "Carry out
         ↪ responsibilities, which include a, b, c, d in
         ↪ accordance with company policies" for 'Employer at
         ↪ all times where time is "Performing Duties Time")
    }
  )
  ("1.98 Changes in duties by employer" ; 'Employer is permitted
     ↪  to "change the employee's assignment, duties and
     ↪ responsibilities and reporting arrangements by the
```

```
    ↪ Employer in its sole discretion without causing
    ↪ termination of this agreement" for 'Employee at any
    ↪ time repeated where time is "Change Duties By Employer
    ↪ Time")
("1.99 Changes in duties agreement by employee" ; 'Employee is
    ↪  obligated to "agree with his new assignment, duties
    ↪ and responsibilities and reporting arrangements" for '
    ↪ Employer at each "Change Duties By Employer Time" where
    ↪  time is "Change in Duties Acknowledgement Time")


("2. Compensation" ; all where time is "Compensation
    ↪ Fulfillment Time"
{
  ("2.1 Salary Payment" ; 'Employer is obligated to transfer
      ↪ 10000 USD for 'Employee every (0 , 1 , 0) starting at
      ↪  ("Start Time" + (24 , 0 , 0)) and ending at "End
      ↪ Time" where time is "Salary Payment Time")
  ("2.2 Salary Review" ; 'Employer is obligated to "Review
      ↪ Salary in 2.1" for 'Employee every (0 , 0 , 1)
      ↪ starting at ("Start Time" + (29 , 11 , 0)) and ending
      ↪  at "End Time" where time is "Salary Review Time")
})
```

```
("2.97 Salary Deductions" ; 'Employer is permitted to "perform
   ↪  statutory deductions" for 'Employee at each "Salary
   ↪ Payment Time" where time is "Salary Deductions Time")
("2.98 Approved Expense Exercise" ; 'Employee is permitted to
   ↪ "Exercise Approved Expense" for 'Employer at any time
   ↪ repeated where time is "Approved Expense Exercise Time
   ↪ ")
("2.99 Unapproved Expense Exercise" ; 'Employee is prohibited
   ↪ from "Exercise Unapproved Expense" for 'Employer at any
   ↪  time repeated where time is "Unapproved Expense
   ↪ Exercise Time")

("3. Vacation" ; 'Employee is permitted to "take a 4 week
   ↪ paid vacation" for 'Employer every (0 , 0 , 1) starting
   ↪  at ("Start Time" + (29 , 11 , 0)) and ending at "End
   ↪ Time" where time is "Vacation Time")
("4. Benefits" ; 'Employer is obligated to "provide health
   ↪ plan" for 'Employee at "Start Time" where time is "
   ↪ Health Benefits Time")
("5. Probation" ; 'Employer is permitted to "terminate
   ↪ employment without cause" for 'Employee between "Start
   ↪ Time" and ("Start Time" + (0 , 3 , 0)) where time is "
```

```
    ↪ Employment Termination Time")
("6. Performance Reviews" ; 'Employer is obligated to "Give
    ↪ performance review and Discuss it" for 'Employee every
    ↪ (0 , 0 , 1) starting at ("Start Time" + (29 , 11 , 0))
    ↪ and ending at "End Time" where time is "Employee
    ↪ Performance Review Time")
("7. Termination" ; any where time is "Employment Termination
    ↪  Time"
{
  ("7.1 Employee Termination of Agreement" ; 'Employee is
      ↪ permitted to "Terminate in lieu of two weeks notice"
      ↪ for 'Employer at any time where time is "Employee
      ↪ Termination of Agreement Time")
  ("7.2 Employer Termination With Cause" ; 'Employer is
      ↪ permitted to "terminate this Agreement and the
      ↪ 'Employees employment, without notice or payment in
      ↪ lieu of notice, for sufficient cause" for 'Employee
      ↪ at any time where time is "Employer Termination With
      ↪ Cause Time")
  ("7.3 Termination Without Cause Clause" ; all where time is
      ↪ "Termination Without Cause Fulfillment Time" {
    ("7.3.1 Employer Termination Without Cause" ; 'Employer is
```

```
        ↪  permitted to "terminate this Agreement and the
        ↪ 'Employees employment without sufficient cause" for
        ↪ 'Employee at any time where time is "Employer
        ↪ Termination Without Cause Time")
    ("7.3.2 Employer Termination Severance Package" ; '
        ↪ Employer is obligated to "pay an amount as required
        ↪ by the Employment Standards Act 2000 or other such
        ↪ legislation as may be in effect at the time of
        ↪ termination and it shall constitute the employees
        ↪ entire entitlement arising from said termination"
        ↪ for 'Employee at "Employer Termination Without Cause
        ↪  Time" where time is "Employer Termination
        ↪ Entitlement Payment Time")
  })
})
("7.99 Returning employer property after termination" ; '
    ↪ Employee is obligated to "any property of the employer"
    ↪  for 'Employer at "Employment Termination Time" where
    ↪ time is "Return Employer Property Time")
("8. Non Competition" ; all where time is "Non Competition
    ↪ Fulfillment Time"
{
```

```
  ("8.1 No Hiring previous employees" ; 'Employee is
     ↪ prohibited from "hiring current company employees"
     ↪ for 'Employer after "Employment Termination Time"
     ↪ where time is "Hiring Current Company Employees Time
     ↪ ")
  ("8.2 Soliciting business from company clients" ; 'Employee
     ↪ is prohibited from "soliciting business from company
     ↪ clients" for 'Employer within (0 , 6 , 0) of "
     ↪ Termination Clause Fulfillment Time" where time is "
     ↪ Hiring Current Company Employees Time")
})
("9. Laws" ; all where time is "Laws Fulfillment Time"
{
  ("9.1 Laws Acknowledgement by Employee" ; 'Employee is
     ↪ obligated to "Acknowledge that this entire agreement
     ↪ is governed by the laws of the province of Ontario"
     ↪ for 'Employer at "Start Time" where time is "Laws
     ↪ Acknowledgement by Employee Time")
  ("9.2 Laws Acknowledgement by Employer" ; 'Employer is
     ↪ obligated to "Acknowledge that this entire agreement
     ↪ is governed by the laws of the province of Ontario"
     ↪ for 'Employee at "Start Time" where time is "Laws
```

```
    ↪ Acknowledgement by Employer Time")
})
("10. Independent Legal Advice" ; all where time is "
   ↪ Independent Legal Advice Fulfillment Time"
{
  ("10.1 Independent Legal Advice Acknowledgement by Employee"
     ↪  ; 'Employee is obligated to "Acknowledge that they
     ↪ were given reasonable opportunity to obtain
     ↪ independent legal advice with respect to this
     ↪ agreement" for 'Employer at "Start Time" where time
     ↪ is "Independent Legal Advice Acknowledgement by
     ↪ Employee Time")
  ("10.2 Independent Legal Advice Choices" ; any where time is
     ↪  "Independent Legal Advice Choices Fulfillment Time"
  {
    ("10.2.1 Independent Legal Advice Choice 1" ; 'Employee is
       ↪  obligated to "Acknowledge that The Employee has had
       ↪  such independent legal advice prior to executing
       ↪ this agreement" for 'Employer at "Independent Legal
       ↪ Advice Acknowledgement by Employee Time" where time
       ↪ is "Legal Advice Choice 1 Time")
    ("10.2.2 Independent Legal Advice Choice 2" ; 'Employee is
```

```
      ↪  obligated to "Acknowledge that they have willingly
      ↪ chosen not to obtain such advice and to execute this
      ↪  agreement without having obtained such advice." for
      ↪  'Employer at "Independent Legal Advice
      ↪ Acknowledgement by Employee Time" where time is "
      ↪ Legal Advice Choice 2 Time")
  })
})
("11. Entire Agreement" ; all where time is "Entire Agreement
   ↪  Fulfillment Time"
{
  ("11.1 Entire Agreement Employee" ; 'Employee is obligated
     ↪ to "Acknowledge that contains the entire agreement
     ↪ between the parties" for 'Employer at "Start Time"
     ↪ where time is "Entire Agreement Employee Time")
  ("11.2 Entire Agreement Employer" ; 'Employer is obligated
     ↪ to "Acknowledge that contains the entire agreement
     ↪ between the parties" for 'Employee at "Start Time"
     ↪ where time is "Entire Agreement Employer Time")
})
("12. Severability" ; all where time is "Severability
   ↪ Fulfillment Time"
```

```
    {
      ("12.1 Severability Employee" ; 'Employee is obligated to "
         ↪ Acknowledge that in the event any article or part
         ↪ thereof of this agreement is held to be unenforceable
         ↪  or invalid then said article or part shall be struck
         ↪  and all remaining provision shall remain in full
         ↪ force and effect" for 'Employer at "Start Time" where
         ↪  time is "Entire Agreement Employee Time")
      ("12.2 Severability Employer" ; 'Employer is obligated to "
         ↪ Acknowledge that in the event any article or part
         ↪ thereof of this agreement is held to be unenforceable
         ↪  or invalid then said article or part shall be struck
         ↪  and all remaining provision shall remain in full
         ↪ force and effect" for 'Employee at "Start Time" where
         ↪  time is "Entire Agreement Employer Time")
    })
  }
}
```

5.3.4.2 Simulation in Maude

We simulate this example in two different environments:

- Case where all obligations and prohibitions are fulfilled except ones about

termination and none of the permissions is exercised.

- Case where all obligations and prohibitions are breached and none of the permissions is exercised.

Case where all obligations and prohibitions are fulfilled except ones about termination and none of the permissions is exercised. The environment for this case is very interesting because of the number of events needed for triggering fulfillment for all the clauses. Most of the clauses are simple clauses that require a single event to trigger their fulfillment, however, some of them require a lot more than that. One clause in particular that requires a lot of events is "1.1 Duties". This clause requires at least a single event for each single day of the contract. We present the summarized environment in Listing 5.16. It contains 365+ events and each of those translates to a variable in the state. The clauses about termination are all breached/ not exercised, while all the rest are fulfilled. A portion of the resulting state is shown in Figure 5.21. This simple case highlights how big a state can get, even with relatively simple contracts and clauses. In general, the "at all times" time constraint is a very expensive constraint processing wise, so it must be used with care.

One agreement within the list of agreements that are fulfilled is the "Prohibition Clause" agreement which is not in the original contract, but it is inherited from the Non Disclosure Agreement contract we defined earlier. Notice how SlanC deals with the inherited clause as if it were defined in this contract and not in its parent.

Listing 5.16: Environment for the case where all obligations and prohibitions are fulfilled except ones about termination and none of the permissions is exercised.

```
(("provide a Senior Software Engineer Position" by 'Employer for '
   ↪ Employee) @ (1 , 1 , 2018) )
(("provide health plan" by 'Employer for 'Employee) @ (1 , 1 ,
   ↪ 2018) )
(("Acknowledge that this entire agreement is governed by the laws
   ↪ of the province of Ontario" by 'Employer for 'Employee) @ (1
   ↪  , 1 , 2018) )
(("Acknowledge that this entire agreement is governed by the laws
   ↪ of the province of Ontario" by 'Employee for 'Employer) @ (1
   ↪  , 1 , 2018) )
(("Acknowledge that they were given reasonable opportunity to
   ↪ obtain independent legal advice with respect to this
   ↪ agreement" by 'Employee for 'Employer) @ (1 , 1 , 2018) )
(("Acknowledge that The Employee has had such independent legal
   ↪ advice prior to executing this agreement" by 'Employee for '
   ↪ Employer) @ (1 , 1 , 2018) )
(("Acknowledge that they have willingly chosen not to obtain such
   ↪ advice and to execute this agreement without having obtained
   ↪  such advice." by 'Employee for 'Employer) @ (1 , 1 , 2018)
   ↪ )
```

```
(("Acknowledge that contains the entire agreement between the
   ↪ parties" by 'Employee for 'Employer) @ (1 , 1 , 2018) )
(("Acknowledge that contains the entire agreement between the
   ↪ parties" by 'Employer for 'Employee) @ (1 , 1 , 2018) )
(("Acknowledge that in the event any article or part thereof of
   ↪ this agreement is held to be unenforceable or invalid then
   ↪ said article or part shall be struck and all remaining
   ↪ provision shall remain in full force and effect" by '
   ↪ Employee for 'Employer) @ (1 , 1 , 2018) )
(("Acknowledge that in the event any article or part thereof of
   ↪ this agreement is held to be unenforceable or invalid then
   ↪ said article or part shall be struck and all remaining
   ↪ provision shall remain in full force and effect" by '
   ↪ Employer for 'Employee) @ (1 , 1 , 2018) )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (1 , 1 , 2018) )
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (25 , 1 , 2018) )
((transfer 10000 USD by 'Employer for 'Employee) @ (25 , 1 , 2018)
```

```
   ↪ in fulfillment of "2.1 Salary Payment" )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (26 , 1 , 2018) )
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (25 , 2 , 2018) )
((transfer 10000 USD by 'Employer for 'Employee) @ (25 , 2 , 2018)
   ↪ in fulfillment of "2.1 Salary Payment" )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (26 , 2 , 2018) )
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (25 , 3 , 2018) )
((transfer 10000 USD by 'Employer for 'Employee) @ (25 , 3 , 2018)
   ↪ in fulfillment of "2.1 Salary Payment" )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (26 , 3 , 2018) )
```

```
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (25 , 4 , 2018) )
((transfer 10000 USD by 'Employer for 'Employee) @ (25 , 4 , 2018)
   ↪ in fulfillment of "2.1 Salary Payment" )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (26 , 4 , 2018) )
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (30 , 4 , 2018) )
(("change the employee's assignment, duties and responsibilities
   ↪ and reporting arrangements by the Employer in its sole
   ↪ discretion without causing termination of this agreement" by
   ↪  'Employer for 'Employee) @ (1 , 5 , 2018) )
(("agree with his new assignment, duties and responsibilities and
   ↪ reporting arrangements" by 'Employee for 'Employer) @ (1 , 5
   ↪  , 2018) )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
```

```
   ↪ ) @ (1 , 5 , 2018) )
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (25 , 5 , 2018) )
((transfer 10000 USD by 'Employer for 'Employee) @ (25 , 5 , 2018)
   ↪ in fulfillment of "2.1 Salary Payment" )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (26 , 5 , 2018) )
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (25 , 6 , 2018) )
((transfer 10000 USD by 'Employer for 'Employee) @ (25 , 6 , 2018)
   ↪ in fulfillment of "2.1 Salary Payment" )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (26 , 6 , 2018) )
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
```

```
   ↪ ) @ (25 , 7 , 2018) )
((transfer 10000 USD by 'Employer for 'Employee) @ (25 , 7 , 2018)
   ↪ in fulfillment of "2.1 Salary Payment" )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (26 , 7 , 2018) )
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (25 , 8 , 2018) )
((transfer 10000 USD by 'Employer for 'Employee) @ (25 , 8 , 2018)
   ↪ in fulfillment of "2.1 Salary Payment" )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (26 , 8 , 2018) )
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (24 , 9 , 2018) )
((transfer 10000 USD by 'Employer for 'Employee) @ (25 , 9 , 2018)
   ↪ in fulfillment of "2.1 Salary Payment" )
(("Carry out responsibilities, which include a, b, c, d in
```

```
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (25 , 9 , 2018) )
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (24 , 10 , 2018) )
((transfer 10000 USD by 'Employer for 'Employee) @ (25 , 10 , 2018)
   ↪  in fulfillment of "2.1 Salary Payment" )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (25 , 10 , 2018) )
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (24 , 11 , 2018) )
((transfer 10000 USD by 'Employer for 'Employee) @ (25 , 11 , 2018)
   ↪  in fulfillment of "2.1 Salary Payment" )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (25 , 11 , 2018) )
....
(("Carry out responsibilities, which include a, b, c, d in
```

```
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (24 , 12 , 2018) )
((transfer 10000 USD by 'Employer for 'Employee) @ (25 , 12 , 2018)
   ↪  in fulfillment of "2.1 Salary Payment" )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (25 , 12 , 2018) )
...
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (30 , 12 , 2018) )
(("Give performance review and Discuss it" by 'Employer for '
   ↪ Employee) @ (30 , 12 , 2018) )
(("Review Salary in 2.1" by 'Employer for 'Employee) @ (30 , 12 ,
   ↪ 2018) )
(("Carry out responsibilities, which include a, b, c, d in
   ↪ accordance with company policies" by 'Employee for 'Employer
   ↪ ) @ (31 , 12 , 2018) )
(("xcv" by 'Employee for 'Employer) @ (1 , 1 , 2019))
```

Case where all obligations and prohibitions are breached and none of the permissions is exercised. In this case we simulate the contract under an environment that causes all the clauses with in it to be breached. One such

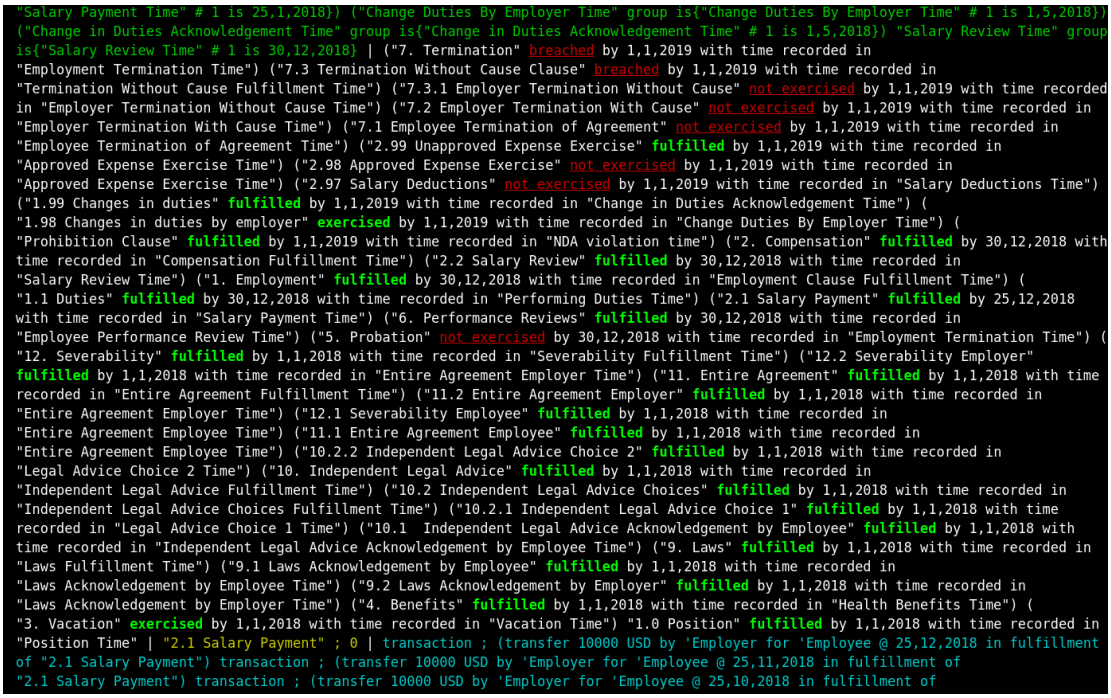

Figure 5.21: Result of the case where all obligations and prohibitions are fulfilled except ones about termination and none of the permissions is exercised.

environment is shown in Listing 5.17, where the employee violates the non compete agreement and discloses info about their work and exercises an unapproved expense. All of the aforementioned events violate the prohibitions which span the duration of the contract. The result is shown in Figure 5.22, where only one agreement is fulfilled, which is an obligation to accept change in duties by the employee, however there were no changes to begin with, i.e fulfilled.

Listing 5.17: Environment for the case where all obligations and prohibitions are fulfilled except ones about termination and none of the permissions is exercised.

```
(("hiring current company employees" by 'Employee for 'Employer) @
    ↪ (1 , 5 , 2018) )
```

```
(("disclose info about the work done" by 'Employee for 'Employer) @
    ↪  (1 , 5 , 2018) )
(("Exercise Unapproved Expense" by 'Employee for 'Employer) @ (9 ,
    ↪ 9 , 2018) )
(("xcv" by 'Employee for 'Employer) @ (1 , 1 , 2019))
```

```
result ContractState: noassets | ("Start Time" is 1,1,2018) ("End Time" is 31,12,2018) ("NDA violation time" group is{"NDA violation time"
    # 1 is 1,5,2018}) "Unapproved Expense Exercise Time" group is{"Unapproved Expense Exercise Time" # 1 is 9,9,2018} | ("7. Termination"
    breached by 1,1,2019 with time recorded in "Employment Termination Time") ("7.3 Termination Without Cause Clause" breached by 1,1,2019
    with time recorded in "Termination Without Cause Fulfillment Time") ("7.3.1 Employer Termination Without Cause" not_exercised by 1,1,
    2019 with time recorded in "Employer Termination Without Cause Time") ("7.2 Employer Termination With Cause" not_exercised by 1,1,2019
    with time recorded in "Employer Termination With Cause Time") ("7.1 Employee Termination of Agreement" not_exercised by 1,1,2019 with
    time recorded in "Employee Termination of Agreement Time") ("6. Performance Reviews" breached by 1,1,2019 with time recorded in
    "Employee Performance Review Time") ("3. Vacation" not_exercised by 1,1,2019 with time recorded in "Vacation Time") (
    "2.99 Unapproved Expense Exercise" breached by 1,1,2019 with time recorded in "Unapproved Expense Exercise Time") (
    "2.98 Approved Expense Exercise" not_exercised by 1,1,2019 with time recorded in "Approved Expense Exercise Time") (
    "2.97 Salary Deductions" not_exercised by 1,1,2019 with time recorded in "Salary Deductions Time") ("2. Compensation" breached by 1,1,
    2019 with time recorded in "Compensation Fulfillment Time") ("2.2 Salary Review" breached by 1,1,2019 with time recorded in
    "Salary Review Time") ("2.1 Salary Payment" breached by 1,1,2019 with time recorded in "Salary Payment Time") (
    "1.99 Changes in duties agreement by employee" fulfilled by 1,1,2019 with time recorded in "Change in Duties Acknowledgement Time") (
    "1.98 Changes in duties by employer" not_exercised by 1,1,2019 with time recorded in "Change Duties By Employer Time") ("1.1 Duties"
    breached by 1,1,2019 with time recorded in "Performing Duties Time") ("Prohibition Clause" breached by 1,1,2019 with time recorded in
    "NDA violation time") ("12. Severability" breached by 1,5,2018 with time recorded in "Severability Fulfillment Time") (
    "12.2 Severability Employer" breached by 1,5,2018 with time recorded in "Entire Agreement Employer Time") (
    "12.1 Severability Employee" breached by 1,5,2018 with time recorded in "Entire Agreement Employee Time") ("11. Entire Agreement"
    breached by 1,5,2018 with time recorded in "Entire Agreement Fulfillment Time") ("11.2 Entire Agreement Employer" breached by 1,5,2018
    with time recorded in "Entire Agreement Employer Time") ("11.1 Entire Agreement Employee" breached by 1,5,2018 with time recorded in
    "Entire Agreement Employee Time") ("10. Independent Legal Advice" breached by 1,5,2018 with time recorded in
    "Independent Legal Advice Fulfillment Time") ("10.1  Independent Legal Advice Acknowledgement by Employee" breached by 1,5,2018 with
    time recorded in "Independent Legal Advice Acknowledgement by Employee Time") ("9. Laws" breached by 1,5,2018 with time recorded in
    "Laws Fulfillment Time") ("9.2 Laws Acknowledgement by Employer" breached by 1,5,2018 with time recorded in
    "Laws Acknowledgement by Employer Time") ("9.1 Laws Acknowledgement by Employee" breached by 1,5,2018 with time recorded in
    "Laws Acknowledgement by Employee Time") ("5. Probation" not_exercised by 1,5,2018 with time recorded in
    "Employment Termination Time") ("4. Benefits" breached by 1,5,2018 with time recorded in "Health Benefits Time") ("1. Employment"
    breached by 1,5,2018 with time recorded in "Employment Clause Fulfillment Time") "1.0 Position" breached by 1,5,2018 with time
    recorded in "Position Time" | nopairs | notransactions
```

Figure 5.22: Result of the case where all obligations and prohibitions are breached and none of the permissions are exercised.

5.4 Threats to validity

There are several threats to the validity of the results presented in the previous two sections, as follows:

- Selection of contracts: The criteria for the selection of contracts was mostly subjective with the singular goal of choosing ”different” contracts. We tried

to reduce this subjectiveness as much as possible by choosing contracts from different domains.

- Translation of contracts: Translating contracts to SlanC was fully dependent on my understanding of the contract, which is limited at best. This is to be expected because of my limited understanding of law.

- Test cases: Coverage might be inadequate because these test cases were not constructed with a testers mentality.

CHAPTER 6

EVALUATION OF SLANC

This chapter presents an evaluation of SlanC, form the following 2 perspectives: expressiveness, usability and performance, which are presented in sections 6.1 **??** and 6.3, respectively.

6.1 Evaluation of expressiveness

In this section we evaluate the expressiveness of SlanC based on a set of requirements for contract languages presented by [31], which include the following expressivity related requirements:

- modelling of contract participants: This is needed to enable specifying who is making the promise and to whom and it is made and also crucial for blame assignment in case of violation. SlanC supports modelling of contract participants in two major ways, namely, defining all the parties in the preamble and using their IDs in all the directed modalities(obligation etc..) to explicitly state who makes the commitment and to whom. Furthermore,

the ID of the contract participant can be used to make explicit an asset belongs to whom.

- (conditional) commitments, i.e., obligations, permissions, and prohibitions: These modalities are at the heart of all contracts, because they describe different types of commitments. SlanC supports all these modalities, through different types of agreements and rules.

- absolute and relative temporal constraints: SlanC support both types of temporal constraints. Absolute temporal contraints can be given in the form constant dates to the different types of constraints supported by the language. Relative temporal constraints can also be expressed where the fulfillment of any clause can be constrained relative to the fulfillment time of another clause.

- basic arithmetic: This is also partially supported in SlanC, because in place arithmetic can be used explicitly on time only. Furthermore, most of the arithmetic pertaining to assets and asset transfers are abstracted from the users and is dynamically tracked by the semantics of SlanC. This dynamic tracking is still limited in the current version of SlanC.

- contrary-to-duty (reparation clauses): These are fully supported by SlanC; in fact, SlanC goes a step further by supporting expression of repeated violations and their reparation. For example, in SlanC it is possible to construct two agreements such that the second one specifies a reparation for each violation of the first one, where the first one may be violated many times.

- potentially infinite and repetitive contracts: This feature is needed by some auto renewed contracts. It is not supported by SlanC

- blame assignment: This is closely related to (R10) and it refers to the notion that when there is a violation the model must be able to assign blame, i.e who violated what clause. SlanC fully supports blame assignment.

Table 6.1, shows a table where all the above requirements and their specification with a comparison between SlanC and 3 other languages from the literature. The choice of these languages to compare SlanC to was made based on the fact that these languages are the most recent development for in their class. The developments in each of these classes was presented in the Chapter 2.

Table 6.1: Requirements for contract languages and their specification with a comparison between SlanC and 3 representative contract languages ([1], [2], [3])

General Requirements	**Specific Requirements**	**SlanC**	**[1]**	**[2]**	**[3]**
Expressing contract start time	a contract must have a field for its start time	Yes	No	Yes	Yes
Expressing contract end time	a contract must have a field for its end time	Yes	No	Yes	Yes
Expressing contract participants	A participant must have a name field	Yes	No	No	Yes

	A participant must have an address field	Yes	No	No	Yes
	A participant must have an id field	Yes	Yes	No	Yes
	A contract must have a participants field that stores a list of participants	Yes	No	No	Yes
Expressing assets	an asset must have a code field that is used to refer to the asset	Yes	No	Yes	No
	an asset must have a supply field that specifies the available amount of that asset	Yes	No	No	No
	an asset must have an owner field that which participant owns the asset by their id	Yes	No	No	No
Expressing actions	must enable expressing generic (string) undirected actions.	Yes	Yes	Yes	Yes
	must enable expressing generic (string) directed actions, that specify a performer and a beneficiary.	Yes	Yes	Yes	Yes

	must enable expressing directed asset transfer actions	Yes	No	No	Yes
Expressing obligations	must enable expressing the benefiting party	Yes	Yes	No	Yes
	must enable expressing promising party	Yes	Yes	No	Yes
	must enable use of all types of actions described in the "expressing actions" requirement.	Yes	Yes	No	Yes
	must enable use of absolute time constraints to limit the scope of fulfillment to a certain timeframe as shown in the "absolute time constraints" requirement.	Yes	Yes	Yes	Yes
	must enable use of relative time constraints to limit the scope of fulfillment to a certain timeframe as shown in the "absolute time constraints" requirement.	Yes	Yes	Yes	Yes
Expressing permissions	must enable expressing the benefitting party	Yes	No	No	Yes

	must enable expressing promising party	Yes	No	No	Yes
	must enable use of all types of actions described in the "expressing actions" requirement.	Yes	No	No	Yes
	must enable use of absolute time constraints to limit the scope of fulfillment to a certain timeframe as shown in the "absolute time constraints" requirement.	Yes	No	No	Yes
	must enable use of relative time constraints to limit the scope of fulfillment to a certain timeframe as shown in the "absolute time constraints" requirement.	Yes	No	No	Yes
Expressing prohibitions	must enable expressing the benefiting party	Yes	Yes	No	Yes
	must enable expressing promising party	Yes	Yes	No	Yes
	must enable use of all types of actions described in the "expressing actions" requirement.	Yes	Yes	No	Yes

	must enable use of absolute time constraints to limit the scope of fulfillment to a certain timeframe as shown in the "absolute time constraints" requirement.	Yes	Yes	Yes	Yes
	must enable use of relative time constraints to limit the scope of fulfillment to a certain timeframe as shown in the "absolute time constraints" requirement.	Yes	Yes	Yes	Yes
Expressing conditional commitments	must enable expressing conditions that trigger commitments when they are satisfied	Yes	Yes	Yes	Yes
Expressing absolute time constraints	must enable expressing temporal constraints using static time (e.g 5 / 5 / 2018)	Yes	Yes	Yes	Yes
Expressing relative time constraints	must enable expressing temporal constraints relative to some known reference point in time like the start time of the contract.	Yes	Yes	Yes	Yes
Expressing reparation clauses	must enable expressing reparation of breached obligation	No	Yes	Yes	Yes

	must enable expressing reparation of breached prohibition	Yes	Yes	Yes	Yes
basic arithmetic	must enable the use of basic time arithmetic on times and time references.	Yes	Yes	Yes	Yes
Expressing infinite contracts	must enable expressing contracts that do not have an end time.	No	No	No	Yes
Expressing repetitive contracts	must enable expressing contracts that auto renew if they are not cancelled. a renewal means resetting all relevant conditions and commitments to their original state before the beginning of the current term in at the start of the new term.	No	No	No	Yes
Expressing blame assignment	must enable expressing when a condition is fulfilled	Yes	Yes	No	Yes
	must enable expressing when a condition is breached	Yes	Yes	No	Yes
	must enable expressing when a commitment is fulfilled	Yes	Yes	No	Yes

	must enable expressing when a commitment is breached	Yes	Yes	No	Yes
	must enable expressing the participant that violated a commitment	Yes	Yes	No	Yes
	must enable expressing the participant that violated a condition	Yes	Yes	No	Yes
	must enable expressing the participant that fulfilled a commitment	Yes	Yes	No	Yes
	must enable expressing the participant that fulfilled a condition	Yes	Yes	No	Yes

6.2 Evaluation of usability

In this section we present a preliminary evaluation of the usability of SlanC from the viewpoint of language users using two critera. Then, we compare SlanC against other languages in the literature with respect to usability based on those criteria. These criteria and their metrics are defined in Table 6.2.

The evaluation based on both criteria are presented for SlanC and other representative languages from the literature are presented in Table 6.3. Furthermore, the reason for choosing these 3 concepts only is that they are core concepts that

Table 6.2: Criteria and metrics for usability evaluation

Criteria	Description	Metric	Justification
Closeness of Mapping between actual concepts and their specifications	Quantify the difference between specifications written in formal languages and their actual representation.	Ratio of number of components used per concept in the formal representation	Closeness of mapping has a direct impact on the expressiveness and accessibility. It gives a general indication of whether a formal language can capture a all aspects of a given real world concept or not and whether it would be simpler or more difficult thank the real world representation.
Coverage of domain concepts.	The number of concepts from the domain that can be represented by the formal language.	Number of covered concepts in the formal language from known domain concepts.	Basic measure of expressive power in terms of domain concepts, which only considers general and well known domain concepts.

Table 6.3: Number components needed to specify the concepts of obligations, prohibitions and permissions in SlanC and other langauges

Domain Concept	**Number of components in the real world representation**	**Number of components in SlanC**	**Number of components in [1]**	**Number of components in [2]**	**Number of components in [3]**
Obligation	5	6	6	2	6
Prohibition	5	6	6	2	6
Permission	5	6	infinity (inexpressible)	2	6

exist in almost all types of contracts.

Table 6.3, shows the number components needed to specify the concepts of obligations, prohibitions and permissions in the real world and in 4 formal languages for writing contracts including SlanC. The five components that are needed to express each of these are: the subject, beneficiary, action(promise), time constraint and essence of the concept. For example, a simple obligation in its real world representation might look like the following:

```
the buyer is must transfer 100 USD to the seller on May 5th 2018
```

This has the 5 components that we mentioned earlier: the subject(buyer), the beneficiary(seller), action(transfer 100 USD), the time constraint(on May 5th 2018) and finally the essence of the obligation(the keyword **must** which differentiates obligations from other domain concepts). Furthermore, when translating this obligation to each language, we obtain the following:

```
SlanC: 'Buyer is obligated to transfer 100 USD for 'Seller at (5 /
   ↪ 5 / 2018) where time is "transfer time"
[1]: O"((Transfer "Stock, transfer(stock), {seller},{buyer}), 5]).
[2]: transfer (5,5,2018) 100 USD
[3]: <Obligation subject="Buyer" beneficiary="Seller">
       <Rel>Pay100USDOnMay5th2018</Rel>
       <Var>PaymentDate</Var>
     </Obligation>
```

Notice that in all languages except [2], an obligation is expressed using 6 components while the real representation of the obligation had only 5. Moreover, this criteria did not consider **how** these languages expressed an obligation, rather, it considered only what was used to express it. In other words, regardless of the actual syntax of the language a user has to learn about an extra component that was not **explicitly** present in the real world representation of an obligation. Therefore, one can conclude from this that these languages are more complex than natural language in terms of expressing obligations. The same conclusions could be drawn about the other two concepts(i.e permissions and prohibitions) because they have the same number of components.

Using this line of reasoning, one might conclude that the language in [2] is simpler than all other languages, even natural language!. This conclusion holds, however, this is done at the expense of expressivity, as shown in the example above there is no mention of the beneficiary and the subject which are implicitly

understood based on whom holds the contract. Furthermore, the essence of the concept is not described but it can be inferred from the word ”transfer” which could be arbitrarily defined to be anything. In general, the lower the number of components the simpler the language, however, when the number of components in the formal representation dips below the number used in the real representation then there is an adverse impact on expressivity.

The same table (6.3) indicates that most of these languages have the expressive power to express the core components of contracts. The only language which is lacking in the respect is [1], because it can not express permissions.

6.3 Evaluation of Performance of SlanC and its Maude specification

This section presents the performance evaluation of SlanC, which we conduct based on data obtained from the Maude environment. We start by presenting information about the environment where the tests were conducted in section 6.3.1. Then we present the results from evaluating each of the real world examples in section 6.3.2. Finally, we present a ranking of the major types of constructs of SlanC based on their impact on performance and validate that ranking through the Maude specification.

6.3.1 Environment and Metrics of the Performance Evaluation

Knowledge about the environment where the evaluation was conducted is necessary to gauge the relative change in performance with change in the environment. We conducted all the tests inside a virtual machine running Ubuntu 16.04 64Bit which was managed by VirtualBox Virtual Machine Manager. This virtual environment was running on a laptop with the following specifications:

- CPU: Intel Core i7 7400MQ @ 2.4GHz
- RAM: 16GB DDR3
- Host Operating System: Windows 10 64Bit
- Guest Operating System: Ubuntu 16.04 64Bit
- Guest OS was given full access to the CPU and its cores.
- Guest OS was given 8GB of RAM.

As far as the performance evaluation is concerned these specifications are not important because we do not use time to measure the performance of the system. These specifications can be used to approximate how much more processing power is needed to improve the performance by a certain factor.

The unit we use for measuring the performance of SlanCs Maude specification is a `rewrite`. A `rewrite` is an atomic operation within Maude that reduces a given term into another desired term. The current implementation of Maude

does millions of rewrites per second on the average for equational specifications, depending largely on the particular machine that it runs on. The reason for using rewrites and not the actual time is that the number of rewrites will never change given that the equation specification and its arguments do not change. In other words, a `rewrite` is a machine independent measure of performance, besides the time could always be calculated by multiplying the number of rewrites that were done to compute the output by the number of rewrites the Maude system could do on the given machine.

6.3.2 Results of the performance evaluation of the specification

We present in this section an evaluation of the performance as observed for each of the cases that we presented the section 5.3.

Table 6.4 shows the numbers of rewrites and events for each of the cases for all the real world examples we discussed in section **??**. Examining the table by column we get the number of rewrites and events for a single contract in all cases, where we observe that the number of rewrites grows linearly with the number of events with small differences that can be attributed to the types of clauses that the actual events trigger. For example, an event that triggers an agreement will definitely have less number of rewrites than another agreement that triggers a rule, because of the obvious difference in complexity. Furthermore, when we examine outliers in the table we see that in some cells there is a single event hundreds to

thousands of rewrites depending on the size of the contract because Maude must first parse it before evaluation. Another type of outlier is the value in the first case of the employment agreement (61 million rewrites) and there are 2 main factors that contributed to this abnormally large value:

- Number of clauses within the contract, contributes linearly to the number of rewrites and as it grows the number of rewrites grows.

- Use of the ”at all times” time constraint: This time constraint in particular is very expensive, because it causes the agreement/condition that contains it to be fulfilled only when there is at least one event that satisfies the action coinciding with this time constraint(i.e same agreement or condition) at each increment of time between the start time and the end time of that contract. For example, in the employment agreement contract the agreement that used this time constraint had an action about performing the duties and in order to fulfill that agreement alone we had to include 365 ”performing duties” events, which are distributed over all days between start time and end time. This number would obviously change depending on the length of the contract, so if the contract was a 2 year contract then there would 730 events. The reason why this time constraint is so expensive can be attributed to it triggering the use of the function `count-increments` which adds tens of thousands of rewrites per call, which is done with every event that satisfies action coinciding with this time constraint. Furthermore, the pure functional nature of Maude is the root of this problem, because all

Contract/Case	Parking Space Agreement	Indemnity Agreement	Bill of Sale	Employment Agreement
Case 1	**33/251913**	**2/1649**	**11/12608**	**394/61306558**
Case 2	**3/25570**	**6/6065**	**1/1528**	**4/603215**
Case 3	**27/215764**	**1/856**	**11/16305**	**-**
Case 4	**-**	**5/5177**	**11/7596**	**-**

Table 6.4: The number of rewrites done with the number of events in each case for each real world example of a contract discussed in section 5.3. Each cell shows "number of events / number of rewrites".

the calls to `count-increments` in the same contract with the "at all times" constraint will have the same result, because the start and end times for the contract do not change, therefore a single evaluation would suffice.

For reference, the case with the largest number of executed rewrites will took about 16 seconds to execute on the aforementioned hardware. This means that the Maude interpreter does 3 million+ rewrites per second, which is running inside a virtual machine and could potentially be significantly improved if the Maude interpreter was running directly on the host operating system.

These performance numbers are dependant on the prototype of SlanC, therefore, even the slightest change in the implementation could possibly result in major differences in performance in some cases. For example, the evaluation of an agreement requires checking its time constraint and its action against the time and action of the event, and depending on the order these operations are performed could differ. In order to illustrate this, we use the following agreement as an example:

```
'A is obligated to "say thank you" for 'B at (1 , 1 , 2018) where
```

Order	Checking actions first	Checking time constraints first
Number of rewrites (sequential)	103	52
Number of rewrites (parallel)	108	

Table 6.5: Change in number of rewrites when the order of the conditions checking actions and time constraints is changed.

```
↪ time is "thank you time"
```

Which we evaluate in the following environment:

```
"some event" by 'A for 'B @ (2 , 1 , 2018)
```

We do this evaluation in two variations of the specification and compare the number of rewrites. These two variations switch the order of evaluation of time constraints and actions. Table 6.5 shows the number of rewrites in both cases, which are obviously the best and worst case scenario, because both conditions do not apply and therefore whichever is evaluated first can be used to conclude the final result which is false.

Performing less number of rewrites might not always guarantee the best throughput, because sometimes parallel evaluation of both conditions could result in better throughput with higher number for rewrites. In that case, swapping the order of the conditions will not matter, because they will both be executed in parallel.

6.3.3 Relative performance of the constructs of SlanC

In this section rank the 3 main constructs in SlanC based on their impact on the performance of the violation detection semantics. These 3 constructs are agreements, rules and containers.

Intuitively, we can rank these constructs from most expensive to least expensive as follows:

- Agreements: An agreement is the simplest type of clause in SlanC it does not have any other clauses encapsulated within it.
- Containers: A container that contains at least one agreement is obviously more expensive than an agreement because it requires the processing of the agreement and the processing of the container. The reason there must be at least one agreement within the container is that a container is meaningless without having at least one child and an agreement is the simplest type of child.
- Rules: The only difference between a rule and a container is a condition, therefore, a rules is at least more expensive than a container by the amount of processing needed by a condition.

We verify this through the developed Maude specification by examining the number of rewrites produced by each type of construct. A rewrite is an atomic operation within Maude that is not machine dependant. It only depends on the Maude engine, the SlanC specification and the input.

We start by examining the number of rewrites produced by the following agreement:

```
'A is obligated to "say thank you" for 'B at (1 / 1 / 2018) where
   ↪ time is "thank you time"
```

Which we evaluate in the following environment:

```
"say thank you" by 'A for 'B @ (1 / 1 / 2018)
```

Then we examine the number of rewrites produced by a container encapsulating the previous agreement in a container, as follows:

```
all where time is "all time"{
'A is obligated to "say thank you" for 'B at (1 / 1 / 2018) where
   ↪ time is "thank you time"
}
```

Which we evaluate in the same environment as shown above. Finally, we examine the number of rewrite produced by encapsulating the previous container in a rule, as follows:

```
if "say hi" by 'B for 'A at (1 / 1 / 2018) where time is "hi time"
then all where time is "all time"{
'A is obligated to "say thank you" for 'B at (1 / 1 / 2018) where
   ↪ time is "thank you time"
}
```

Which we simulate in the following environment:

Construct type	Agreements	Containers	Rules
Number of rewrites	55	112	566

Table 6.6: Relative performance in terms of rewrites for agreements, containers and rules

```
"say hi" by 'B for 'A @ (1 / 1 / 2018)

"say thank you" by 'A for 'B @ (1 / 1 / 2018)
```

Clearly, the aforementioned constructs increase in complexity as we transition from agreements to containers to rules. This complexity also translates in terms of impact on performance, as shown by the results in Table 6.6.

6.4 Limitations of SlanC

This section presents the major limitations of SlanC, as follows:

- No way of specifying a contract type: SlanC does not have a way of specifying a contract type, which could enable defining the semantics at a more granular level. In other words, instead of having one semantic definition for all types of contracts, different semantics could be developed for different types of contracts, which could possibly have the same syntax.

- Cannot Express infinite or repetitive contracts: These types of contracts have an initial end time that is renewed automatically, if none of the participants of the contracts cancel. Currently, SlanC does not support this feature, mainly because the notions of contract termination and renewal are

not understood by the language, i.e a clause about termination or renewal is specified like any other clause.

- SlanC clauses are static: Currently, SlanC only supports static clauses that do not change at all. For example, consider a modified parking lease agreement that has its lease value increasing by %5 every year.

- SlanC lack active clauses: The current version of SlanC only supports passive clauses. An active clause change other clauses while a passive clause does not. A good example of this is contracts present in insurance agreements (e.g car insurance) where the amount of paid yearly insurance could increase in response to fulfilling or breaching other agreements.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this thesis we presented SlanC, a domain specific language for writing contracts. SlanC is uniquely characterized by having syntax that is well structured, yet closely resembles the variations of natural language that are used to describe contracts, which we anticipate will make the language more accessible to general audiences. Furthermore, we developed a formal executable model of it in Maude and specified and simulated several small examples and real world contracts. Moreover, we evaluated the expressiveness of SlanC found out that it is expressive enough to capture basic contracts, based on requirements identified by [15] for contract languages. Also, usability of SlanC . Finally, we evaluated the relative performance of major SlanC constructs using its specification and we found out that agreements have the least performance overhead, while rules have the highest.

While SlanC has many of the features needed by a language for writing general

contracts, it still cannot be used to express general contracts effectively. This is in part due to our limited understanding of contracts and law in general. However, while developing the language and its semantics we gained better understanding of contracts which will help in directing our efforts to improve its expressiveness and usability. In improving SlanCs expressiveness, we believe that we must improve the way actions are expressed, and the best way to achieve that is through better understanding the types of actions that exist in contracts. Therefore, it is crucial to conduct a survey of common types of actions that exist in contracts. Currently, we only have two types of actions, namely, an arbitrary action and an asset transfer action, but there are still many other type that are common to most contracts e.g contract termination. In addition to the type of action, action evolution must also be considered because we encountered some contracts where the actions evolve/devolve over time. For example, in the case of a rental agreement where the rent increases by 5 percent each year.

In improving its usability, we believe that SlanC must be implemented on the blockchain, because while its Maude specification is useful in understanding and verifying the language, it unusable in a real legal environment, because it is not secure. Blockchain and distributed ledger technology is appealing because they offer the security, transparency and resilience necessary for the sensitive nature of contracts.

APPENDIX A

THE SLANC MAUDE SPECIFICATION

```
fmod CLAUSES is
pr STRING .
sorts Clause ClauseID OverrideClause ClauseList .
subsort OverrideClause < Clause < ClauseList .
subsort String < ClauseID .
op nilclause : -> Clause .
op noclauses : -> ClauseList .
op _ _ : ClauseList ClauseList -> ClauseList [id: noclauses assoc ]
   ↪  .
op override _ : Clause -> OverrideClause .
endfm
fmod TIME is
pr NAT .
pr BOOL .
sort Time TimeError TimeOperation .
subsort Time < TimeError .
op rem(_,_) : Nat Nat -> Nat .
op quo(_,_) : Nat Nat -> Nat .
op min(_,_) : Time Time -> Time .
op max(_,_) : Time Time -> Time .
op lt(_,_) : Time Time -> Bool .
op lte(_,_) : Time Time -> Bool .
op max-days-month(_) : Nat -> Nat .
op time-error : -> TimeError .
ops undefined-time undefinable-time : -> Time .
```

```
op _,_,_ : Nat Nat Nat -> Time .
op noop : -> TimeOperation .
op - : -> TimeOperation .
op -- : -> TimeOperation .
op + : -> TimeOperation .
op ++ : -> TimeOperation .
op _ _ _ : Time TimeOperation Time -> Time .
op validate-time(_) : Time -> TimeError .
op fix-zeros(_) : Time -> Time .
vars D1 D2 M1 M2 Y1 Y2 : Nat .
vars T1 T2 : Time .
eq rem(D1,D2) = D1 rem D2 .
eq quo(D1,D2) = D1 quo D2 .
eq validate-time(undefinable-time) = undefinable-time .
eq validate-time(undefined-time) = undefined-time .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 31 or M1 > 12
   ↪ .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 31 and M1 == 1
   ↪ .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 28 and M1 == 2
   ↪ .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 31 and M1 == 3
   ↪ .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 30 and M1 == 4
   ↪ .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 31 and M1 == 5
   ↪ .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 30 and M1 == 6
   ↪ .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 31 and M1 == 7
   ↪ .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 31 and M1 == 8
   ↪ .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 30 and M1 == 9
   ↪ .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 31 and M1 ==
   ↪ 10 .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 30 and M1 ==
   ↪ 11 .
ceq validate-time(D1 , M1 , Y1) = time-error if D1 > 31 and M1 ==
   ↪ 12 .
eq validate-time(D1 , M1 , Y1) = D1 , M1 , Y1 [owise] .
ceq T1 ++ T2 = undefinable-time if T1 == undefinable-time or T2 ==
   ↪ undefinable-time .
```

```
ceq T1 ++ T2 = undefined-time if T1 == undefined-time or T2 ==
   ↪ undefined-time .
ceq (D1 , M1 , Y1) ++ (D2 , M2 , Y2) = time-error if validate-time(
   ↪ D1 , M1 , Y1) == time-error or validate-time(D2 , M2 , Y2)
   ↪ == time-error .

eq (D1 , M1 , Y1) ++ (D2 , M2 , Y2) =
fix-zeros(rem(D1 + D2,max-days-month(M1) + 1) ,
rem(M1 + M2 + quo(D1 + D2,max-days-month(M1) + 1),13) , Y1 + Y2 +
   ↪ quo(M1 + M2 + quo(D1 + D2,max-days-month(M1) + 1 ),13)) .
ceq fix-zeros((D1 , M1 , Y1)) = (D1 + 1) , (M1 + 1) , Y1 if D1 == 0
   ↪  and M1 == 0 .
ceq fix-zeros((D1 , M1 , Y1)) = 1 , M1 , Y1 if D1 == 0 .
ceq fix-zeros((D1 , M1 , Y1)) = (D1) , (M1 + 1) , Y1 if M1 == 0 .
eq fix-zeros((D1 , M1 , Y1)) = (D1) , (M1) , Y1 .
eq max-days-month(0) = 31 .
eq max-days-month(1) = 31 .
eq max-days-month(2) = 28 .
eq max-days-month(3) = 31 .
eq max-days-month(4) = 30 .
eq max-days-month(5) = 31 .
eq max-days-month(6) = 30 .
eq max-days-month(7) = 31 .
eq max-days-month(8) = 31 .
eq max-days-month(9) = 30 .
eq max-days-month(10) = 31 .
eq max-days-month(11) = 30 .
eq max-days-month(12) = 31 .
ceq lt(T1,T2) = false if T1 == undefined-time or T1 == undefinable-
   ↪ time or T2 == undefined-time or T2 == undefinable-time .
eq lt((D1 , M1 , Y1), (D2 , M2 , Y2)) = Y1 < Y2 or (Y1 == Y2 and M1
   ↪  < M2) or (Y1 == Y1 and M1 == M2 and D1 < D2) .
ceq lte(T1,T2) = false if T1 == undefined-time or T1 == undefinable
   ↪ -time or T2 == undefined-time or T2 == undefinable-time .
eq lte(T1,T2) = lt(T1,T2) or T1 == T2 .
eq min(T1,T2) = if lte(T1,T2) then T1 else T2 fi .
eq max(T1,T2) = if lte(T1,T2) then T2 else T1 fi .
endfm
fmod ASSETS is
pr STRING .
pr INT .
pr QID .
sorts Asset AssetCode AssetName AssetList PartyID .
subsort String < AssetName .
subsort Qid < PartyID .
```

```
subsort Asset < AssetList .
ops USD SAR EUR : -> AssetCode .
op {name:_,code:_,supply:_,owner:_} : String AssetCode Int PartyID
   ↪ -> Asset .
op noassets : -> AssetList .
op _ _ : AssetList AssetList -> AssetList [assoc id: noassets] .
endfm
fmod PARTIES is
pr ASSETS .
sorts Address Party PartyList .
subsort Party < PartyList .
subsort String < Address .
op noaddress : -> Address .
op {id:_,name:_,description:_,address:_,assets:[_]} : PartyID
   ↪ String String Address AssetList -> Party .
op {id:_,name:_,description:_,address:_} : PartyID String String
   ↪ Address -> Party .
op noparties : -> PartyList .
op _ _ : PartyList PartyList -> PartyList [id: noparties assoc] .
endfm
fmod VARIABLES is
pr TIME .
pr PARTIES .
pr QID .
pr CLAUSES .
sorts Variable VariableList VariableGroup VariableDeclaration
   ↪ VariableName VariableValue VariableUpdateStatement
   ↪ TimeArithmatic .
subsort String < VariableName < TimeArithmatic .
subsort VariableUpdateStatement < Clause .
subsort Variable < VariableList .
subsort VariableGroup < Variable .
op novariablename : -> VariableName .
op _ # _ : String Nat -> VariableName .
op where time is _ : VariableName -> VariableDeclaration .
op novariable : -> Variable .
op _ is _ : VariableName Time -> Variable .
op _ is _ : VariableName String -> Variable .
op _ is _ : VariableName PartyID -> Variable .
op _ group is undefined-group : VariableName -> VariableGroup .
op _ group is {_} : VariableName VariableList -> VariableGroup .
op _ _ _ : VariableName TimeOperation Time -> TimeArithmatic .
op _ ; add _ to _ : ClauseID Time VariableName ->
   ↪ VariableUpdateStatement .
```

```
op _ ; subtract _ from _ : ClauseID Time VariableName ->
   ↪ VariableUpdateStatement .
op novariables : -> VariableList .
op _ _ : VariableList VariableList -> VariableList [assoc id:
   ↪ novariables] .
vars VL VL2 : VariableList .
vars VN1 VN2 : VariableName .
vars VG1 VG2 : VariableGroup .
vars V V1 V2 : Variable .
var T1 T2 : Time .
var N1 N2 : Nat .
vars S1 : String .
var PID : PartyID .
vars CID1 CID2 : ClauseID .
op get-variable-list(_) : VariableGroup -> VariableList .
op get-variable-name(_) : TimeArithmatic -> VariableName .
op get-added-value(_) : TimeArithmatic -> Time .
op get-time-operation(_) : TimeArithmatic -> TimeOperation .
op count-variables(_) : VariableList -> Nat .
op get-single-variable-time-value(_,_) : VariableName VariableList
   ↪ -> Time .
op get-variable-group(_,_) : VariableName VariableList ->
   ↪ VariableGroup .
op evaluate-time-arithmatic(_,_) : TimeArithmatic VariableList ->
   ↪ Time .
op heads-time(_) : VariableList -> Time .
op update-variable-group(_,_,_) : VariableName Variable
   ↪ VariableGroup -> VariableGroup .
op get-time-at(_,_) : Nat VariableList -> Time .
op update-variable-with-offest(_,_) : VariableUpdateStatement
   ↪ VariableList -> VariableList .
eq get-time-at(N1,novariables) = undefined-time .
eq get-time-at(0,VL (VN1 # N2 is T1)) = T1 .
eq get-time-at(N1,VL V) = get-time-at(N1 - 1,VL) .
eq get-time-operation( VN1 ) = ++ .
eq get-time-operation( VN1 noop T1 ) = ++ .
eq get-time-operation( VN1 + T1 ) = ++ .
eq get-time-operation( VN1 - T1 ) = -- .
eq get-variable-name( VN1 ) = VN1 .
eq get-variable-name( VN1 noop T1 ) = VN1 .
eq get-variable-name( VN1 + T1 ) = VN1 .
eq get-variable-name( VN1 - T1 ) = VN1 .
eq get-added-value( VN1 ) = (0 , 0 , 0) .
eq get-added-value( VN1 noop T1 ) = T1 .
eq get-added-value( VN1 + T1 ) = T1 .
```

```
eq get-added-value( VN1 - T1 ) = T1 .
eq get-single-variable-time-value(VN1, novariables) = undefined-
   ↪ time .
eq get-single-variable-time-value(VN1, VG1 VL) = get-single-
   ↪ variable-time-value(VN1, VL) .
eq get-single-variable-time-value(VN1, (VN2 is S1) VL) = get-single
   ↪ -variable-time-value(VN1, VL) .
eq get-single-variable-time-value(VN1, (VN2 is PID) VL) = get-
   ↪ single-variable-time-value(VN1, VL) .
eq get-single-variable-time-value(VN1, (VN2 is T1) VL) = if VN1 ==
   ↪ VN2 then T1 else get-single-variable-time-value(VN1, VL) fi
   ↪ .
eq get-variable-group(VN1,novariables) = VN1 group is undefined-
   ↪ group .
eq get-variable-group(VN1, (VN2 group is {VL2}) VL) = if VN1 == VN2
   ↪  then VN2 group is {VL2} else get-variable-group(VN1,VL) fi
   ↪ .
eq get-variable-group(VN1, V VL) = get-variable-group(VN1,VL) .
eq update-variable-group(VN1,V,novariables) = VN1 group is {V} .
ceq update-variable-group(VN1,V,(VN2 group is {V2 VL2}) VL) = (VN2
   ↪ group is {V V2 VL2}) VL if VN1 == VN2 and not(V == V2) .
ceq update-variable-group(VN1,V,(VN2 group is {V2 VL2}) VL) = (VN2
   ↪ group is {V2 VL2}) VL if VN1 == VN2 .
eq update-variable-group(VN1,V, V1 VL) = V1 update-variable-group(
   ↪ VN1,V,VL) .
eq heads-time(novariables) = undefined-time .
eq heads-time((VN1 is T1) VL) = T1 .
eq heads-time((VN1 # N1 is T1) VL) = T1 .
eq get-variable-list(VN1 group is undefined-group) = novariables .
eq get-variable-list(VN1 group is {VL}) = VL .
eq evaluate-time-arithmatic(VN1,VL) = get-single-variable-time-
   ↪ value(VN1,VL) .
eq evaluate-time-arithmatic(VN1 noop T1,VL) = get-single-variable-
   ↪ time-value(VN1,VL) .
eq evaluate-time-arithmatic(VN1 + T1,VL) = get-single-variable-time
   ↪ -value(VN1,VL) ++ T1 .
eq evaluate-time-arithmatic(VN1 - T1,VL) = get-single-variable-time
   ↪ -value(VN1,VL) -- T1 .
eq count-variables(novariables) = 0 .
eq count-variables(V VL) = 1 + count-variables(VL) .
eq update-variable-with-offest( CID1 ; add T1 to VN1, novariables)
   ↪ = novariables .
eq update-variable-with-offest( CID1 ; add T1 to VN1, VG1 VL) = VG1
   ↪  update-variable-with-offest( CID1 ; add T1 to VN1,VL) .
```

```
eq update-variable-with-offest( CID1 ; add T1 to VN1, (VN2 is T2)
   ↪ VL) = if VN1 == VN2 then (VN2 is (T2 ++ T1)) VL else (VN2 is
   ↪  T2) update-variable-with-offest( CID1 ; add T1 to VN1,VL)
   ↪ fi .
endfm
fmod CONSTRAINTS is
pr TIME .
pr VARIABLES .
pr BOOL .
sorts TimeConstraint .
op at any time : -> TimeConstraint .
op at any time repeated : -> TimeConstraint .
op at all times : -> TimeConstraint .
op at _ : Time -> TimeConstraint .
op before _ : Time -> TimeConstraint .
op after _ : Time -> TimeConstraint .
op between _ and _ : Time Time -> TimeConstraint .
op within _ of _ : Time Time -> TimeConstraint .
op every _ starting at _ and ending at _ : Time Time Time ->
   ↪ TimeConstraint .
op at _ : TimeArithmatic -> TimeConstraint .
op before _ : TimeArithmatic -> TimeConstraint .
op after _ : TimeArithmatic -> TimeConstraint .
op between _ and _ : TimeArithmatic TimeArithmatic ->
   ↪ TimeConstraint .
op between _ and _ : Time TimeArithmatic -> TimeConstraint .
op between _ and _ : TimeArithmatic Time -> TimeConstraint .
op within _ of _ : Time TimeArithmatic -> TimeConstraint .
op every _ starting at _ and ending at _ : Time TimeArithmatic
   ↪ TimeArithmatic -> TimeConstraint .
op every _ starting at _ and ending at _ : Time Time TimeArithmatic
   ↪  -> TimeConstraint .
op every _ starting at _ and ending at _ : Time TimeArithmatic Time
   ↪  -> TimeConstraint .
op at each _ : TimeArithmatic -> TimeConstraint .
op within _ of each _ : Time TimeArithmatic -> TimeConstraint .
vars T1 T2 T3 T4 : Time .
vars N1 N2 : Nat .
vars V1 V2 V3 V4 : Variable .
vars VN1 VN2 VN3 VN4 : VariableName .
vars TA1 TA2 TA3 : TimeArithmatic .
vars VL1 VL2 : VariableList .
vars TC1 TC2 : TimeConstraint .
op satisfies(_,_,_,_) : Time VariableName TimeConstraint
   ↪ VariableList -> Bool .
```

```
op exceeds(_,_,_) : Time TimeConstraint VariableList -> Bool .
op update-with-constraint(_,_,_,_) : VariableName Time
   ↪ TimeConstraint VariableList -> VariableList .
op make-new-group-variable(_,_,_) : VariableName Time VariableList
   ↪ -> Variable .
op count-increments(_,_) : TimeConstraint VariableList -> Nat .
op is-arbitrary-increment(_,_,_,_) : Time Time Time Time -> Bool .
op get-increment-number(_,_,_,_,_) : Time Time Time Time Nat -> Nat
   ↪  .
ceq is-arbitrary-increment(T1,T2,T3,T4) = true if (T1 == T3) or (T1
   ↪  == T4) .
ceq is-arbitrary-increment(T1,T2,T3,T4) = false if lte(T4,T3) .
eq is-arbitrary-increment(T1,T2,T3,T4) = is-arbitrary-increment(T1,
   ↪ T2,T3 ++ T2,T4) .
ceq get-increment-number(T1,T2,T3,T4,N1) = N1 if (T1 == T3) .
ceq get-increment-number(T1,T2,T3,T4,N1) = N1 if lte(T4,T3) .
eq get-increment-number(T1,T2,T3,T4,N1) = get-increment-number(T1,
   ↪ T2,T3 ++ T2,T4,(N1 + 1)) .
eq satisfies(T1,VN2,at T2,VL1) = T1 == T2 .
eq satisfies(T1,VN2,after T2,VL1) = lt(T2,T1) .
eq satisfies(T1,VN2,before T2,VL1) = lt(T1,T2) .
eq satisfies(T1,VN2,between T2 and T3,VL1) = lte(T1,T3) and lte(T2,
   ↪ T1) .
eq satisfies(T1,VN2,within T2 of T3,VL1) = satisfies(T1,VN2,between
   ↪  T3 and (T3 ++ T2),VL1) .
eq satisfies(T1,VN2,every T2 starting at T3 and ending at T4,VL1) =
   ↪  lte(T1,T4) and lte(T3,T1) and (T1 == (heads-time(get-
   ↪ variable-list(get-variable-group(VN2,VL1))) ++ T2) or T1 ==
   ↪ T3) .
eq satisfies(T1,VN2,at TA1,VL1) = satisfies(T1,VN2,at evaluate-time
   ↪ -arithmatic(TA1,VL1),VL1) .
eq satisfies(T1,VN2,before TA1,VL1) = satisfies(T1,VN2,before
   ↪ evaluate-time-arithmatic(TA1,VL1),VL1) .
eq satisfies(T1,VN2,after TA1,VL1) = satisfies(T1,VN2,after
   ↪ evaluate-time-arithmatic(TA1,VL1),VL1) .
eq satisfies(T1,VN2,between TA1 and TA2,VL1) = satisfies(T1,VN2,
   ↪ between evaluate-time-arithmatic(TA1,VL1) and evaluate-time-
   ↪ arithmatic(TA2,VL1),VL1) .
eq satisfies(T1,VN2,between T2 and TA2,VL1) = satisfies(T1,VN2,
   ↪ between T2 and evaluate-time-arithmatic(TA2,VL1),VL1) .
eq satisfies(T1,VN2,between TA1 and T3,VL1) = satisfies(T1,VN2,
   ↪ between evaluate-time-arithmatic(TA1,VL1) and T3,VL1) .
eq satisfies(T1,VN2,within T2 of TA1,VL1) = satisfies(T1,VN2,within
   ↪  T2 of evaluate-time-arithmatic(TA1,VL1),VL1) .
```

```
eq satisfies(T1,VN2,every T2 starting at TA1 and ending at TA2,VL1)
   ↪  =
satisfies(T1,VN2,every T2 starting at evaluate-time-arithmatic(TA1,
   ↪ VL1) and ending at evaluate-time-arithmatic(TA2,VL1),VL1) .
eq satisfies(T1,VN2,every T2 starting at TA1 and ending at T3,VL1)
   ↪ =
satisfies(T1,VN2,every T2 starting at evaluate-time-arithmatic(TA1,
   ↪ VL1) and ending at T3,VL1) .
eq satisfies(T1,VN2,every T2 starting at T3 and ending at TA2,VL1)
   ↪ =
satisfies(T1,VN2,every T2 starting at T3 and ending at evaluate-
   ↪ time-arithmatic(TA2,VL1),VL1) .
eq satisfies(T1,VN2,at each TA1,VL1) =
satisfies(T1,VN2,at
(get-time-at(count-variables(get-variable-list(get-variable-group(
   ↪ VN2,VL1))),
get-variable-list(get-variable-group( get-variable-name(TA1) ,VL1)
   ↪ )) get-time-operation(TA1) get-added-value(TA1) ) ,VL1) .
eq satisfies(T1,VN2,within T2 of each TA1,VL1) =
satisfies(T1,VN2,within T2 of (get-time-at(count-variables(get-
   ↪ variable-list(get-variable-group(VN2,VL1))),
get-variable-list(get-variable-group( get-variable-name(TA1) ,VL1)
   ↪ )) get-time-operation(TA1) get-added-value(TA1)) ,VL1) .
eq satisfies(T1,VN2,at any time repeated,VL1) = not(validate-time(
   ↪ T1) == time-error) and lte(T1,get-single-variable-time-value
   ↪ ("End Time",VL1)) .
eq satisfies(T1,VN2,at any time,VL1) = not(validate-time(T1) ==
   ↪ time-error) and lte(T1,get-single-variable-time-value("End
   ↪ Time",VL1)) .
eq satisfies(T1,VN2,at all times,VL1) = not(validate-time(T1) ==
   ↪ time-error) and lte(T1,get-single-variable-time-value("End
   ↪ Time",VL1)) .
eq update-with-constraint(VN1,T1,every T2 starting at T3 and ending
   ↪  at T4,VL1) = update-variable-group(VN1,VN1 # get-increment-
   ↪ number(T1,T2,T3,T4,1) is T1 ,VL1) .
eq update-with-constraint(VN1,T1,every T2 starting at T1 and ending
   ↪  at TA2,VL1) = update-with-constraint(VN1,T1,every T2
   ↪ starting at T1 and ending at evaluate-time-arithmatic(TA2,
   ↪ VL1),VL1) .
eq update-with-constraint(VN1,T1,every T2 starting at TA1 and
   ↪ ending at T2,VL1) = update-with-constraint(VN1,T1,every T2
   ↪ starting at evaluate-time-arithmatic(TA1,VL1) and ending at
   ↪ T2,VL1) .
eq update-with-constraint(VN1,T1,every T2 starting at TA1 and
   ↪ ending at TA2,VL1) = update-with-constraint(VN1,T1,every T2
```

```
   ↪ starting at evaluate-time-arithmatic(TA1,VL1) and ending at
   ↪ evaluate-time-arithmatic(TA2,VL1),VL1) .
eq update-with-constraint(VN1,T1,at all times,VL1) =
update-variable-group(VN1, make-new-group-variable(VN1,T1,VL1) ,VL1
   ↪ ) .
eq update-with-constraint(VN1,T1,at any time repeated,VL1) =
update-variable-group(VN1, make-new-group-variable(VN1,T1,VL1) ,VL1
   ↪ ) .
eq update-with-constraint(VN1,T1,at each TA1,VL1) = update-variable
   ↪ -group(VN1,make-new-group-variable(VN1,T1,VL1) ,VL1) .
eq update-with-constraint(VN1,T1,within T2 of each TA1,VL1) =
   ↪ update-variable-group(VN1,make-new-group-variable(VN1,T1,VL1
   ↪ ) ,VL1) .
eq update-with-constraint(VN1,T1, TC1,VL1) = (VN1 is T1) VL1 .
eq make-new-group-variable(VN1,T1,VL1) = VN1 # (count-variables(get
   ↪ -variable-list(get-variable-group(VN1,VL1))) + 1) is T1 .
eq count-increments(at each TA1,VL1) = count-variables(get-variable
   ↪ -list(get-variable-group(get-variable-name(TA1),VL1))) .
ceq count-increments(every T1 starting at T2 and ending at T3,VL1)
   ↪ = 0 if lte(T3,T2) .
eq count-increments(every T1 starting at T2 and ending at T3,VL1) =
   ↪  1 + count-increments(every T1 starting at (T2 ++ T1) and
   ↪ ending at T3,VL1) .
eq count-increments(every T1 starting at TA1 and ending at TA2,VL1)
   ↪  = count-increments(every T1 starting at evaluate-time-
   ↪ arithmatic(TA1,VL1) and ending at evaluate-time-arithmatic(
   ↪ TA2,VL1),VL1) .
eq count-increments(at all times,VL1) = count-increments(every (1 ,
   ↪  0 , 0) starting at "Start Time" and ending at "End Time",
   ↪ VL1) .
eq count-increments(at any time repeated,VL1) = 1 .
eq count-increments(TC1,VL1) = 1 .
eq exceeds(T1,at T2,VL1) = lt(T2,T1) .
eq exceeds(T1,after T2,VL1) = lt(get-single-variable-time-value("
   ↪ End Time",VL1),T1) .
eq exceeds(T1,before T2,VL1) = lte(T2,T1) .
eq exceeds(T1,between T2 and T3,VL1) = lt(T3,T1) .
eq exceeds(T1,within T2 of T3,VL1) = exceeds(T1,between T3 and (T3
   ↪ ++ T2),VL1) .
eq exceeds(T1,every T2 starting at T3 and ending at T4,VL1) = lt(T4
   ↪ ,T1) .
eq exceeds(T1,at TA1,VL1) = exceeds(T1,at evaluate-time-arithmatic(
   ↪ TA1,VL1),VL1) .
eq exceeds(T1,before TA1,VL1) = exceeds(T1,before evaluate-time-
   ↪ arithmatic(TA1,VL1),VL1) .
```

```
eq exceeds(T1,after TA1,VL1) = exceeds(T1,after evaluate-time-
   ↪ arithmatic(TA1,VL1),VL1) .
eq exceeds(T1,between TA1 and TA2,VL1) = exceeds(T1,between
   ↪ evaluate-time-arithmatic(TA1,VL1) and evaluate-time-
   ↪ arithmatic(TA2,VL1),VL1) .
eq exceeds(T1,between T2 and TA2,VL1) = exceeds(T1,between T2 and
   ↪ evaluate-time-arithmatic(TA2,VL1),VL1) .
eq exceeds(T1,between TA1 and T3,VL1) = exceeds(T1,between evaluate
   ↪ -time-arithmatic(TA1,VL1) and T3,VL1) .
eq exceeds(T1,within T2 of TA1,VL1) = exceeds(T1,within T2 of
   ↪ evaluate-time-arithmatic(TA1,VL1),VL1) .
eq exceeds(T1,every T2 starting at TA1 and ending at TA2,VL1) =
exceeds(T1,every T2 starting at evaluate-time-arithmatic(TA1,VL1)
   ↪ and ending at evaluate-time-arithmatic(TA2,VL1),VL1) .
eq exceeds(T1,every T2 starting at TA1 and ending at T3,VL1) =
exceeds(T1,every T2 starting at evaluate-time-arithmatic(TA1,VL1)
   ↪ and ending at T3,VL1) .
eq exceeds(T1,every T2 starting at T3 and ending at TA2,VL1) =
exceeds(T1,every T2 starting at T3 and ending at evaluate-time-
   ↪ arithmatic(TA2,VL1),VL1) .
eq exceeds(T1,at each TA1,VL1) = lt(get-single-variable-time-value
   ↪ ("End Time",VL1),T1) .
eq exceeds(T1,within T2 of each TA1,VL1) = lt(get-single-variable-
   ↪ time-value("End Time",VL1),T1) .
eq exceeds(T1,at any time repeated,VL1) = lt(get-single-variable-
   ↪ time-value("End Time",VL1),T1) .
eq exceeds(T1,at any time,VL1) = lt(get-single-variable-time-value
   ↪ ("End Time",VL1),T1) .
eq exceeds(T1,at all times,VL1) = lt(get-single-variable-time-value
   ↪ ("End Time",VL1),T1) .
endfm
fmod ACTIONS is
pr STRING .
pr PARTIES .
pr TIME .
pr ASSETS .
pr NAT .
pr INT .
sorts Act TransferAct GenericAct Action DirectedBasicAction
   ↪ DirectedAssetTransferAction .
subsorts GenericAct TransferAct < Act .
subsort String < GenericAct .
subsort DirectedBasicAction DirectedAssetTransferAction < Action .
op act : -> Act .
op transfer _ _ : Nat AssetCode -> TransferAct .
```

```
op do _ : String -> GenericAct .
op nilaction : -> Action .
op _ by _ for _ : GenericAct PartyID PartyID -> DirectedBasicAction
   ↪  .
op _ by _ for _ : TransferAct PartyID PartyID ->
   ↪ DirectedAssetTransferAction .
op get-transfer-amount(_) : TransferAct -> Nat .
op update-asset-values-with-action(_,_) : Action AssetList ->
   ↪ AssetList .
op subtract-from-first-party(_,_) : Action AssetList -> AssetList .
op add-to-second-party(_,_) : Action AssetList -> AssetList .
vars N1 N2 : Nat .
vars I1 I2 : Int .
vars S1 S2 : String .
vars AC1 AC2 : AssetCode .
vars ACT1 ACT2 : Action .
vars PID1 PID2 PID3 : PartyID .
vars AL1 AL2 : AssetList .
vars TACT1 TACT2 : DirectedAssetTransferAction .
vars TRA1 TRA2 : TransferAct .
eq get-transfer-amount(transfer N1 AC1) = N1 .
eq update-asset-values-with-action(TRA1 by PID1 for PID2, AL1) =
   ↪ add-to-second-party(TRA1 by PID1 for PID2,subtract-from-
   ↪ first-party(TRA1 by PID1 for PID2,AL1)) .
eq update-asset-values-with-action(ACT1, AL1) = AL1 .
eq subtract-from-first-party(ACT1,noassets) = noassets .
eq subtract-from-first-party(transfer N1 AC1 by PID1 for PID2, {
   ↪ name: S1 ,code: AC2,supply: I1,owner: PID3} AL1) = if PID1
   ↪ == PID3 and AC1 == AC2
then {name: S1 ,code: AC2,supply: (I1 - N1) ,owner: PID3} AL1
else {name: S1 ,code: AC2,supply: I1,owner: PID3} subtract-from-
   ↪ first-party(transfer N1 AC1 by PID1 for PID2, AL1)
fi .
eq add-to-second-party(ACT1,noassets) = noassets .
eq add-to-second-party(transfer N1 AC1 by PID1 for PID2, {name: S1
   ↪ ,code: AC2,supply: I1,owner: PID3} AL1) = if PID2 == PID3
   ↪ and AC1 == AC2
then {name: S1 ,code: AC2,supply: (I1 + N1) ,owner: PID3} AL1
else {name: S1 ,code: AC2,supply: I1 ,owner: PID3} add-to-second-
   ↪ party(transfer N1 AC1 by PID1 for PID2, AL1)
fi .
endfm
fmod CONDITIONS is
pr ACTIONS .
pr PARTIES .
```

```
pr CONSTRAINTS .
pr BOOL .
sort Condition .
subsort Bool < Condition .
op __ is satisfied : Action TimeConstraint -> Condition .
op __ is not satisfied : Action TimeConstraint -> Condition .
endfm
fmod CONTAINERS is
pr CLAUSES .
pr VARIABLES .
pr CONSTRAINTS .
sorts ClauseContainer TimeConstrainedClauseContainer
   ↪ OverrideTimeConstrainedClauseContainer
   ↪ VanillaClauseContainer IDClauseContainer ContainerSpecifier
   ↪ .
subsort TimeConstrainedClauseContainer VanillaClauseContainer
   ↪ IDClauseContainer < ClauseContainer < Clause .
ops any all sequence parallel : -> ContainerSpecifier .
op _ {_} : ContainerSpecifier ClauseList -> VanillaClauseContainer
   ↪ .
op _ ; _ _ {_} : ClauseID ContainerSpecifier VariableDeclaration
   ↪ ClauseList -> IDClauseContainer .
endfm
fmod AGREEMENTS is
pr ACTIONS .
pr CONSTRAINTS .
pr CLAUSES .
sorts Agreement AgreementType Obligation Permission Prohibition .
subsort Obligation Permission Prohibition < Agreement < Clause .
op _ ; _ is obligated to _ for _ _ _ : ClauseID PartyID Act PartyID
   ↪  TimeConstraint VariableDeclaration -> Obligation .
op _ ; _ is permitted to _ for _ _ _ : ClauseID PartyID Act PartyID
   ↪  TimeConstraint VariableDeclaration -> Permission .
op _ ; _ is not obligated to _ for _ _ _ : ClauseID PartyID Act
   ↪ PartyID TimeConstraint VariableDeclaration -> Permission .
op _ ; _ is prohibited from _ for _ _ _ : ClauseID PartyID Act
   ↪ PartyID TimeConstraint VariableDeclaration -> Prohibition .
endfm
fmod RULES is
pr ACTIONS .
pr CONDITIONS .
pr CONTAINERS .
sorts Rule .
subsort Rule < Clause .
```

```
op _ ; if_ _then _ : ClauseID Condition VariableDeclaration
   ↪ VanillaClauseContainer -> Rule .
endfm
fmod STRING-LIST is
pr STRING .
pr BOOL .
sort StringList .
subsort String < StringList .
op none : -> StringList .
op _ _ : StringList StringList -> StringList [id: none assoc] .
op _in_ : String StringList -> Bool .
vars S1 S2 : String .
var SL : StringList .
eq S1 in none = false .
eq S1 in (S2 SL) = if S1 == S2 then true else S1 in SL fi .
endfm
fmod PREAMBLE is
pr STRING .
pr QID .
pr NAT .
pr PARTIES .
pr TIME .
pr STRING-LIST .
sort Preamble .
op Preamble
{
name: _
start time: _
end time: _
parties: [_]
extends: _
} : String Time Time PartyList StringList -> Preamble .
op Preamble : -> Preamble .
op get-name-field(_) : Preamble -> String .
op get-extends-field(_) : Preamble -> StringList .
op get-start-time-field(_) : Preamble -> Time .
op get-end-time-field(_) : Preamble -> Time .
var S1 : String .
var T1 T2 : Time .
var PL1 : PartyList .
var SL1 : StringList .
eq get-name-field(Preamble
{
name: S1
start time: T1
```

```
end time: T2
parties: [PL1]
extends: SL1
}) = S1 .
eq get-extends-field(Preamble
{
name: S1
start time: T1
end time: T2
parties: [PL1]
extends: SL1
}) = SL1 .
eq get-start-time-field(Preamble
{
name: S1
start time: T1
end time: T2
parties: [PL1]
extends: SL1
}) = T1 .
eq get-end-time-field(Preamble
{
name: S1
start time: T1
end time: T2
parties: [PL1]
extends: SL1
}) = T2 .
endfm
fmod CONTRACTS is
pr PREAMBLE .
pr CLAUSES .
pr CONTAINERS .
pr AGREEMENTS .
pr RULES .
pr PREAMBLE .
sort Contract Body .
op Body {_} : ClauseList -> Body .
op Contract {_ _} : Preamble Body -> Contract .
op Body : -> Body .
op Contract : -> Contract .
op get-contract-name(_) : Contract -> String .
op get-contract-ancestors-names(_) : Contract -> StringList .
var P : Preamble .
var B : Body .
```

```
eq get-contract-name(Contract{P B }) = get-name-field(P) .
eq get-contract-ancestors-names(Contract{P B }) = get-extends-field
   ↪ (P) .
endfm
fmod CORE-FUNCTIONS is
pr ACTIONS .
pr CONTAINERS .
pr AGREEMENTS .
pr RULES .
pr CONDITIONS .
pr BOOL .
op get-id(_) : Clause -> ClauseID .
op get-id(_) : OverrideClause -> ClauseID .
op get-time-constraint(_) : Clause -> TimeConstraint .
op get-variable-name(_) : Clause -> VariableName .
op get-action(_) : Clause -> Action .
op get-condition-id(_) : ClauseID -> ClauseID .
op get-consequent-id(_) : ClauseID -> ClauseID .
op get-condition-vn(_) : VariableName -> VariableName .
op get-consequent-vn(_) : VariableName -> VariableName .
var ID : ClauseID .
var CND : Condition .
var VDC : VariableDeclaration .
var CLC : VanillaClauseContainer .
vars PID1 PID2 : PartyID .
var ACT1 : Act .
vars AC1 : Action .
var CL1 : ClauseList .
var VN1 : VariableName .
var TC : TimeConstraint .
var CNTS : ContainerSpecifier .
eq get-id( ID ; if CND VDC then CLC) = ID .
eq get-id( ID ; PID1 is obligated to ACT1 for PID2 TC VDC) = ID .
eq get-id( ID ; PID1 is not obligated to ACT1 for PID2 TC VDC) = ID
   ↪  .
eq get-id( ID ; PID1 is permitted to ACT1 for PID2 TC VDC) = ID .
eq get-id( ID ; PID1 is prohibited from ACT1 for PID2 TC VDC) = ID
   ↪ .
eq get-id( ID ; CNTS VDC {CL1}) = ID .
eq get-id( override (ID ; if CND VDC then CLC)) = ID .
eq get-id( override (ID ; PID1 is obligated to ACT1 for PID2 TC VDC
   ↪ ) ) = ID .
eq get-id( override (ID ; PID1 is not obligated to ACT1 for PID2 TC
   ↪  VDC)) = ID .
```

```
eq get-id( override (ID ; PID1 is permitted to ACT1 for PID2 TC VDC
   ↪ )) = ID .
eq get-id( override (ID ; PID1 is prohibited from ACT1 for PID2 TC
   ↪ VDC)) = ID .
eq get-id( override (ID ; CNTS VDC {CL1})) = ID .
eq get-time-constraint( ID ; if (AC1) (TC) is satisfied VDC then
   ↪ CLC) = TC .
eq get-time-constraint( ID ; if (AC1) (TC) is not satisfied VDC
   ↪ then CLC) = TC .
eq get-time-constraint( ID ; PID1 is obligated to ACT1 for PID2 TC
   ↪ VDC) = TC .
eq get-time-constraint( ID ; PID1 is not obligated to ACT1 for PID2
   ↪  TC VDC) = TC .
eq get-time-constraint( ID ; PID1 is permitted to ACT1 for PID2 TC
   ↪ VDC) = TC .
eq get-time-constraint( ID ; PID1 is prohibited from ACT1 for PID2
   ↪ TC VDC) = TC .
eq get-time-constraint( ID ; CNTS VDC {CL1}) = at any time .
eq get-variable-name( ID ; if CND where time is VN1 then CLC) = VN1
   ↪  .
eq get-variable-name( ID ; PID1 is obligated to ACT1 for PID2 TC
   ↪ where time is VN1) = VN1 .
eq get-variable-name( ID ; PID1 is not obligated to ACT1 for PID2
   ↪ TC where time is VN1) = VN1 .
eq get-variable-name( ID ; PID1 is permitted to ACT1 for PID2 TC
   ↪ where time is VN1) = VN1 .
eq get-variable-name( ID ; PID1 is prohibited from ACT1 for PID2 TC
   ↪  where time is VN1) = VN1 .
eq get-variable-name( ID ; CNTS where time is VN1 {CL1}) = VN1 .
eq get-action( ID ; if (AC1) (TC) is satisfied VDC then CLC) = AC1
   ↪ .
eq get-action( ID ; if (AC1) (TC) is not satisfied VDC then CLC) =
   ↪ AC1 .
eq get-action( ID ; PID1 is obligated to ACT1 for PID2 TC where
   ↪ time is VN1) = ACT1 by PID1 for PID2 .
eq get-action( ID ; PID1 is not obligated to ACT1 for PID2 TC where
   ↪  time is VN1) = ACT1 by PID1 for PID2 .
eq get-action( ID ; PID1 is permitted to ACT1 for PID2 TC where
   ↪ time is VN1) = ACT1 by PID1 for PID2 .
eq get-action( ID ; PID1 is prohibited from ACT1 for PID2 TC where
   ↪ time is VN1) = ACT1 by PID1 for PID2 .
eq get-action( ID ; CNTS where time is VN1 {CL1}) = nilaction .
eq get-condition-id(ID) = ID + "-CONDITION" .
eq get-consequent-id(ID) = ID + "-CONSEQUENT" .
eq get-condition-vn(VN1) = VN1 .
```

```
eq get-consequent-vn(VN1) = VN1 + "-CONSEQUENT" .
endfm
fmod CONTRACT-EXTENSION-SEMANTICS is
pr CONTRACTS .
pr CLAUSES .
pr TIME .
pr CORE-FUNCTIONS .
pr STRING .
sort ContractList .
subsort Contract < ContractList .
op nocontracts : -> ContractList .
op __ : ContractList ContractList -> ContractList [id: nocontracts
   ↪ assoc] .
op resolve-extensions(_,_) : Contract ContractList -> Contract .
op filter-contract-list-by-names(_,_) : Contract ContractList ->
   ↪ ContractList .
op extend-all(_,_,_) : Contract ContractList ContractList ->
   ↪ Contract .
op extend-single(_,_) : Contract Contract -> Contract .
op extend-preamble(_,_) : Preamble Preamble -> Preamble .
op extend-body(_,_) : Body Body -> Body .
op resolve-overrides(_,_) : ClauseList ClauseList -> ClauseList .
op resolve-override(_,_): Clause ClauseList -> ClauseList .
vars C1 C2 : Contract .
var CL CL2 : ContractList .
var P1 P2 : Preamble .
var B1 B2 : Body .
vars T1 T2 T3 T4 : Time .
vars SL1 SL2 : StringList .
vars S1 S2 : String .
vars PL1 PL2 : PartyList .
vars CS1 CS2 CS3 CS4 : ClauseList .
vars CLA1 CLA2 : Clause .
var O : Obligation .
var PRO : Prohibition .
var PER : Permission .
var RL : Rule .
var CNT : TimeConstrainedClauseContainer .
vars RL1 RL2 : Rule .
var ID : ClauseID .
var CND : Condition .
var TC : TimeConstraint .
var VD : VariableDeclaration .
var CNTS : ContainerSpecifier .
```

```
eq resolve-extensions(C1,CL) = extend-all(C1,filter-contract-list-
   ↪ by-names(C1,CL),CL) .
eq filter-contract-list-by-names(C1,nocontracts) = nocontracts .
eq filter-contract-list-by-names(C1,C2 CL) = if (get-contract-name(
   ↪ C2) in get-contract-ancestors-names(C1)) then C2 filter-
   ↪ contract-list-by-names(C1,CL) else filter-contract-list-by-
   ↪ names(C1,CL) fi .
eq extend-all(C1, nocontracts,CL) = C1 .
eq extend-all(C1,(C2 CL),CL2) = extend-all(extend-single(C1,resolve
   ↪ -extensions(C2,CL2)) ,CL,CL2) .
eq extend-single(C1,Contract) = C1 .
eq extend-single(Contract,C1) = C1 .
eq extend-single(C1,Contract{Preamble Body}) = C1 .
eq extend-single(Contract{Preamble Body},C1) = C1 .
eq extend-single(Contract {P1 B1}, Contract {P2 B2}) = Contract {
   ↪ extend-preamble(P1,P2) extend-body(B1,B2) } .
eq extend-preamble(P1,Preamble) = P1 .
eq extend-preamble(Preamble,P2) = P2 .
eq extend-preamble(
Preamble
{
name: S1
start time: T1
end time: T2
parties: [PL1]
extends: SL1

},
Preamble
{
name: S2
start time: T3
end time: T4
parties: [PL2]
extends: SL2

} ) =
Preamble
{
name: S1
start time: T1
end time: T2
parties: [PL1 PL2]
extends: SL1
} .
```

```
eq extend-body(B1,Body) = B1 .
eq extend-body(Body,B2) = B2 .
eq extend-body(Body {CS1} , Body {CS2} ) = Body {resolve-overrides(
   ↪ CS1,CS2)} .
eq resolve-overrides(CS1, noclauses) = CS1 .
eq resolve-overrides(noclauses,CS2) = CS2 .
eq resolve-overrides(override CLA1 CS1 , CS2) = resolve-overrides(
   ↪ CS1,resolve-override(override CLA1,CS2)) .
eq resolve-overrides(CLA1 CS1 , CS2) = resolve-overrides(CS1,CS2
   ↪ CLA1) .
eq resolve-override(override CLA1,noclauses) = override CLA1 .
ceq resolve-override(override CLA1 ,(ID ; if (CND) VD then CNTS {
   ↪ CS1}) CS2) =
(ID ; if (CND) VD then CNTS {resolve-override(override CLA1,CS1)})
   ↪ resolve-override(override CLA1,CS2) if not ID == get-id(CLA1
   ↪ ) .
ceq resolve-override(override CLA1 ,(ID ; CNTS VD {CS1}) CS2) =
(ID ; CNTS VD {resolve-override(override CLA1,CS1)}) resolve-
   ↪ override(override CLA1,CS2) if not ID == get-id(CLA1) .
eq resolve-override(override CLA1,CLA2 CS2) = if get-id(CLA1) ==
   ↪ get-id(CLA2) then ( CLA1 CS2 ) else CLA2 resolve-override(
   ↪ override CLA1,CS2) fi .
endfm
fmod STATE is
pr TIME .
pr ACTIONS .
pr CLAUSES .
pr AGREEMENTS .
pr RULES .
pr CONTAINERS .
pr CORE-FUNCTIONS .
pr INT .
sorts ContractState GlobalClauseState ClauseState Statement
   ↪ AgreementStatement RuleStatement ConditionStatement
   ↪ ContainerStatement StatementList ClauseStateType
   ↪ ClauseStateSpecifier ClauseAssetPairList ClauseAssetPair
   ↪ Transaction TransactionList .
subsort AgreementStatement ContainerStatement ConditionStatement
   ↪ RuleStatement < Statement .
subsort Statement < StatementList .
subsort ClauseAssetPair < ClauseAssetPairList .
subsort ClauseState < GlobalClauseState .
subsort Transaction < TransactionList .
op (_ ; _) : ClauseID Int -> ClauseAssetPair .
op nopairs : -> ClauseAssetPairList .
```

```
op _ _ : ClauseAssetPairList ClauseAssetPairList ->
   ↪ ClauseAssetPairList [assoc id: nopairs] .
op notransactions : -> TransactionList .
op _ _ : TransactionList TransactionList -> TransactionList [assoc
   ↪ id: notransactions] .
op nilstatement : -> Statement .
op nostatements : -> StatementList .
op _ _ : StatementList StatementList -> StatementList [id:
   ↪ nostatements assoc] .
op noclausestate : -> ClauseState .
op noglobalstate : -> GlobalClauseState .
ops nilstatetype fulfillment-state breach-state : ->
   ↪ ClauseStateType .
op _ _ : GlobalClauseState GlobalClauseState -> GlobalClauseState [
   ↪ id: noglobalstate assoc] .
op _ | _ | _ | _ | _ : AssetList VariableList GlobalClauseState
   ↪ ClauseAssetPairList TransactionList -> ContractState [format
   ↪ (r o g o o o y o c o ) ] .
op nocontractstate : -> ContractState .
op fulfilled : -> ClauseStateSpecifier [ format(g! o) ] .
op breached : -> ClauseStateSpecifier [ format(ru o) ] .
op exercised : -> ClauseStateSpecifier [ format(g! o) ] .
op not exercised : -> ClauseStateSpecifier [ format(ru ru o) ] .
op _ _ by _ with time recorded in _ : ClauseID ClauseStateSpecifier
   ↪  Time VariableName -> ClauseState .
op find-state-of(_,_) : ClauseID GlobalClauseState -> ClauseState .
op find-state-by-vn(_,_) : VariableName GlobalClauseState ->
   ↪ ClauseState .
op update-state-of(_,_) : ClauseState GlobalClauseState ->
   ↪ GlobalClauseState .
op get-clause-state-type(_) : GlobalClauseState -> ClauseStateType
   ↪ .
op get-recorded-time(_) : ClauseState -> Time .
vars CL1 CL2 : Clause .
vars CID1 CID2 : ClauseID .
vars RS1 RS2 : RuleStatement .
vars CNTS1 CNTS2 : ContainerStatement .
vars CNDS1 CNDS2 : ConditionStatement .
vars AGS1 AGS2 : AgreementStatement .
vars SL1 SL2 : StatementList .
vars CLS1 CLS2 : ClauseState .
vars GLS1 GLS2 : GlobalClauseState .
vars R1 R2 : Rule .
vars A1 A2 : Agreement .
vars CND1 CND2 : Condition .
```

```
vars CNT1 CNT2 : ClauseContainer .
var CSS1 CSS2 : ClauseStateSpecifier .
vars T1 T2 T3 T4 : Time .
vars VN1 VN2 : VariableName .
vars CAPL1 CAPL2 : ClauseAssetPairList .
eq find-state-of(CID1 , noglobalstate) = noclausestate .
ceq find-state-of(CID1 , (CID2 CSS1 by T1 with time recorded in VN1
   ↪ ) GLS1) = (CID1 CSS1 by T1 with time recorded in VN1) if
   ↪ CID1 == CID2 .
eq find-state-of(CID1 , (CLS1 GLS1)) = find-state-of(CID1, GLS1) [
   ↪ owise] .
eq find-state-by-vn(novariablename , GLS1) = noclausestate .
eq find-state-by-vn(VN1 , noglobalstate) = noclausestate .
ceq find-state-by-vn(VN1 , (CID1 CSS1 by T1 with time recorded in
   ↪ VN2) GLS1) = (CID1 CSS1 by T1 with time recorded in VN2) if
   ↪ VN1 == VN2 .
eq find-state-by-vn(VN1 , (CLS1 GLS1)) = find-state-of(VN1, GLS1) [
   ↪ owise] .
eq get-recorded-time((CID1 fulfilled by T1 with time recorded in
   ↪ VN1)) = T1 .
eq get-recorded-time((CID1 breached by T1 with time recorded in VN1
   ↪ )) = T1 .
eq get-recorded-time((CID1 exercised by T1 with time recorded in
   ↪ VN1)) = T1 .
eq get-recorded-time((CID1 not exercised by T1 with time recorded
   ↪ in VN1)) = T1 .
eq update-state-of(CLS1, noglobalstate) = CLS1 .
ceq update-state-of((CID1 CSS1 by T1 with time recorded in VN1), (
   ↪ CID2 CSS2 by T2 with time recorded in VN2) GLS1) = (CID1
   ↪ CSS1 by T1 with time recorded in VN1) GLS1 if CID2 == CID1 .
eq update-state-of(CLS2, (CLS1 GLS1)) = CLS1 update-state-of(CLS2,
   ↪ GLS1) [owise] .
eq get-clause-state-type(noclausestate) = nilstatetype .
eq get-clause-state-type((CID1 fulfilled by T1 with time recorded
   ↪ in VN1) ) = fulfillment-state .
eq get-clause-state-type((CID1 breached by T1 with time recorded in
   ↪  VN1) ) = breach-state .
eq get-clause-state-type((CID1 exercised by T1 with time recorded
   ↪ in VN1) ) = fulfillment-state .
eq get-clause-state-type((CID1 not exercised by T1 with time
   ↪ recorded in VN1)) = breach-state .
endfm
fmod ENVIRONMENT is
pr TIME .
pr ACTIONS .
```

```
pr CLAUSES .
pr AGREEMENTS .
pr RULES .
pr CONTAINERS .
pr CORE-FUNCTIONS .
pr STATE .
sort Environment Event .
subsort Event < Environment .
op get-event-time(_) : Event -> Time .
op get-event-action(_) : Event -> Action .
op nilevent @ _ : Time -> Event .
op _ @ _ : DirectedBasicAction Time -> Event .
op _ @ _ in fulfillment of _ : DirectedAssetTransferAction Time
   ↪ ClauseID -> Event .
op noevents : -> Environment .
op _ _ : Environment Environment -> Environment [id: noevents assoc
   ↪ ] .
op transaction ; _ : Event -> Transaction .
vars DACT1 DACT2 : DirectedBasicAction .
vars TACT1 TACT2 : DirectedAssetTransferAction .
vars CID1 CID2 : ClauseID .
vars T1 T2 : Time .
eq get-event-time(DACT1 @ T1) = T1 .
eq get-event-time(TACT1 @ T1 in fulfillment of CID1) = T1 .
eq get-event-action(DACT1 @ T1) = DACT1 .
eq get-event-action(TACT1 @ T1 in fulfillment of CID1) = TACT1 .
endfm
fmod VIOLATION-DETECTION-SEMANTICS is
pr ACTIONS .
pr CONSTRAINTS .
pr CONDITIONS .
pr CLAUSES .
pr CONTAINERS .
pr AGREEMENTS .
pr RULES .
pr PREAMBLE .
pr CONTRACTS .
pr CONTRACT-EXTENSION-SEMANTICS .
pr STATE .
pr ENVIRONMENT .
pr CORE-FUNCTIONS .
op detect-violations(_,_,_) : Contract Environment ContractList ->
   ↪ ContractState .
op detect-violations(_,_,_,_) : Contract Environment AssetList
   ↪ ContractList -> ContractState .
```

```
op compute-contract-state(_,_,_) : Contract Environment
   ↪ ContractState -> ContractState .
op update-state-with(_,_) : Preamble ContractState -> ContractState
   ↪  .
op update-assets-with-event(_,_,_,_,_) : ClauseID Action Event
   ↪ TimeConstraint ContractState -> ContractState .
op update-asset-clause-pairs(_,_,_,_,_,_) : ClauseID TransferAct
   ↪ Action TimeConstraint VariableList ClauseAssetPairList ->
   ↪ ClauseAssetPairList .
op update-transaction-list(_,_) : Event TransactionList ->
   ↪ TransactionList .
op update-body-state-given-environment(_,_,_) : Body Environment
   ↪ ContractState -> ContractState .
op update-body-state-given-event(_,_,_) : Body Event ContractState
   ↪ -> ContractState .
op update-clauselist-state-given-event(_,_,_) : ClauseList Event
   ↪ ContractState -> ContractState .
op update-clause-state-given-event(_,_,_) : Clause Event
   ↪ ContractState -> ContractState .
op update-with-agreement(_,_,_) : Agreement Event ContractState ->
   ↪ ContractState .
op update-with-rule(_,_,_) : Rule Event ContractState ->
   ↪ ContractState .
op update-with-variable-update-statement(_,_,_) :
   ↪ VariableUpdateStatement Event ContractState -> ContractState
   ↪  .
op update-with-rule-special-case(_,_,_) : Rule Event ContractState
   ↪ -> ContractState .
op update-with-container(_,_,_) : ClauseContainer Event
   ↪ ContractState -> ContractState .
op update-with-obligation(_,_,_) : Obligation Event ContractState
   ↪ -> ContractState .
op update-with-prohibition(_,_,_) : Prohibition Event ContractState
   ↪  -> ContractState .
op update-with-permission(_,_,_) : Permission Event ContractState
   ↪ -> ContractState .
op check-fulfilled(_,_,_) : Clause Time ContractState -> Bool .
op check-breached(_,_,_) : Clause Time ContractState -> Bool .
op check-fulfilled(_,_,_,_) : Condition VariableName Time
   ↪ ContractState -> Bool .
op check-breached(_,_,_,_) : Condition VariableName Time
   ↪ ContractState -> Bool .
op is-fulfilled(_,_) : ClauseID GlobalClauseState -> Bool .
op is-breached(_,_) : ClauseID GlobalClauseState -> Bool .
```

```
op check-and-update-fulfillment(_,_,_) : Clause Time ContractState
   ↪ -> ContractState .
op check-and-update-fulfillment(_,_,_,_,_) : ClauseID Condition
   ↪ VariableName Time ContractState -> ContractState .
op container-satisfied(_,_,_) : ContainerSpecifier ClauseList
   ↪ GlobalClauseState -> Bool .
op container-breached(_,_,_) : ContainerSpecifier ClauseList
   ↪ GlobalClauseState -> Bool .
op update-with-container-special(_,_,_) : ClauseContainer Event
   ↪ ContractState -> ContractState .
op is-variable-count-match(_,_,_) : VariableName TimeConstraint
   ↪ VariableList -> Bool .
op is-tc-dependency-fulfilled(_,_,_) : Time TimeConstraint
   ↪ ContractState -> Bool .
op is-action-satisfied(_,_,_,_) : ClauseID Action Event
   ↪ ClauseAssetPairList -> Bool .
var C1 C2 : Contract .
var B1 B2 : Body .
var PL1 PL2 : PartyList .
var SL1 SL2 : StringList .
vars S1 S2 : String .
vars CL1 CL2 : ContractList .
var CS1 : ContractState .
vars CSL1 CSL2 : ClauseList .
vars CLAS1 CLAS2 : Clause .
var ENV : Environment .
var EV : Event .
var P1 : Preamble .
var CID1 CID2 CID3 CID4 : ClauseID .
var PID1 PID2 PID3 PID4 : PartyID .
vars AC1 AC2 AC3 AC4 : Act .
vars ACT1 ACT2 : Action .
vars CND1 CND2 : Condition .
vars TC1 TC2 TC3 TC4 : TimeConstraint .
vars VD1 VD2 VD3 : VariableDeclaration .
vars AG1 AG2 : Agreement .
vars O1 O2 O3 : Obligation .
vars PRO1 PRO2 : Prohibition .
vars PER1 PER2 : Permission .
vars P : Preamble .
vars RL1 RL2 : Rule .
var CNT1 CNT2 : ClauseContainer .
vars T1 T2 T3 T4 : Time .
vars AL1 AL2 : AssetList .
vars GS1 GS2 : GlobalClauseState .
```

```
vars VL1 Vl2 : VariableList .
vars VN1 VN2 : VariableName .
vars GENA1 GENA2 : GenericAct .
vars TRA1 TRA2 : TransferAct .
var ASC1 ASC2 : AssetCode .
var VCC : VanillaClauseContainer .
var CNTS : ContainerSpecifier .
vars CAPL1 CAPL2 : ClauseAssetPairList .
vars I1 I2 : Int .
vars N1 N2 : Nat .
vars TL1 TL2 : TransactionList .
vars DACT1 DACT2 : DirectedBasicAction .
vars TACT1 TACT2 : DirectedAssetTransferAction .
vars VUS1 VUS2 : VariableUpdateStatement .
vars TA1 TA2 : TimeArithmatic .
eq detect-violations(C1,ENV,nocontracts) = compute-contract-state(
   ↪ C1,ENV,(noassets | novariables | noglobalstate | nopairs |
   ↪ notransactions)) .
eq detect-violations(C1,ENV,CL1) = compute-contract-state(resolve-
   ↪ extensions(C1,CL1),ENV,(noassets | novariables |
   ↪ noglobalstate | nopairs | notransactions)) .
eq detect-violations(C1,ENV,AL1,CL1) = compute-contract-state(
   ↪ resolve-extensions(C1,CL1),ENV,(AL1 | novariables |
   ↪ noglobalstate | nopairs | notransactions)) .
eq compute-contract-state(Contract{P B1},ENV,CS1) = update-body-
   ↪ state-given-environment(B1,ENV,update-state-with(P,CS1)) .
eq update-state-with(Preamble
{
name: S2
start time: T3
end time: T4
parties: [PL2]
extends: SL2
},(AL1 | VL1 | GS1 | CAPL1 | TL1)) = (AL1 | ("Start Time" is T3) ("
   ↪ End Time" is T4) VL1 | GS1 | CAPL1 | TL1) .
eq update-body-state-given-environment(B1,noevents,CS1) = CS1 .
eq update-body-state-given-environment(B1,EV ENV,CS1) = update-body
   ↪ -state-given-environment(B1,ENV,update-body-state-given-
   ↪ event(B1,EV,CS1)) .
eq update-body-state-given-event(Body { CSL1 },EV,(AL1 | VL1 | GS1
   ↪ | CAPL1 | TL1)) = update-clauselist-state-given-event(CSL1,
   ↪ EV,(update-asset-values-with-action(get-event-action(EV),
   ↪ AL1) | VL1 | GS1 | CAPL1 | update-transaction-list(EV,TL1) )
   ↪ ) .
eq update-clauselist-state-given-event(noclauses,EV,CS1) = CS1 .
```

```
eq update-clauselist-state-given-event(CLAS1 CSL1,EV,CS1) = update-
   ↪ clauselist-state-given-event(CSL1,EV,update-clause-state-
   ↪ given-event(CLAS1,EV,CS1)) .
eq update-clause-state-given-event(AG1,EV,CS1) = update-with-
   ↪ agreement(AG1,EV, CS1 ) .
eq update-clause-state-given-event(RL1,EV,CS1) = update-with-rule(
   ↪ RL1,EV,CS1) .
eq update-clause-state-given-event(CNT1,EV,CS1) = update-with-
   ↪ container(CNT1,EV,CS1) .
eq update-clause-state-given-event(VUS1,EV,CS1) = update-with-
   ↪ variable-update-statement(VUS1,EV,CS1) .
ceq update-with-variable-update-statement(CID1 ; add T1 to VN1,EV,
   ↪ (AL1 | VL1 | GS1 | CAPL1 | TL1)) = (AL1 | VL1 | GS1 | CAPL1
   ↪ | TL1) if is-fulfilled(CID1,GS1) .
eq update-with-variable-update-statement(CID1 ; add T1 to VN1,EV, (
   ↪ AL1 | VL1 | GS1 | CAPL1 | TL1)) =
(AL1 | update-variable-with-offest( CID1 ; add T1 to VN1,VL1) | (
   ↪ CID1 fulfilled by get-event-time(EV) with time recorded in
   ↪ novariablename) GS1 | CAPL1 | TL1) .
eq update-with-variable-update-statement(VUS1,EV, (AL1 | VL1 | GS1
   ↪ | CAPL1 | TL1) ) = (AL1 | VL1 | GS1 | CAPL1 | TL1) .
ceq update-with-agreement(AG1,EV,(AL1 | VL1 | GS1 | CAPL1 | TL1)) =
   ↪  (AL1 | VL1 | GS1 | CAPL1 | TL1) if is-fulfilled(get-id(AG1)
   ↪ ,GS1) or is-breached(get-id(AG1),GS1) .
eq update-with-agreement(O1,EV,CS1) = update-with-obligation(O1,EV,
   ↪ CS1) .
eq update-with-agreement(PRO1,EV,CS1) = update-with-prohibition(
   ↪ PRO1,EV,CS1) .
eq update-with-agreement(PER1,EV,CS1) = update-with-permission(PER1
   ↪ ,EV,CS1) .
eq update-with-obligation(O1,EV,(AL1 | VL1 | GS1 | CAPL1 | TL1)) =
   ↪ if satisfies(get-event-time(EV),get-variable-name(O1),get-
   ↪ time-constraint(O1),VL1) and is-action-satisfied(get-id(O1),
   ↪ get-action(O1),EV,CAPL1)
then check-and-update-fulfillment( O1,get-event-time(EV),
update-assets-with-event(get-id(O1),get-action(O1),EV,get-time-
   ↪ constraint(O1),
(
AL1 |
update-with-constraint(get-variable-name(O1),get-event-time(EV),get
   ↪ -time-constraint(O1),VL1) |
GS1 | CAPL1 | TL1
)
))
```

```
else check-and-update-fulfillment( O1,get-event-time(EV), (AL1 |
    ↪ VL1 | GS1 | CAPL1 | TL1))
fi .
eq update-with-permission(PER1,EV,(AL1 | VL1 | GS1 | CAPL1 | TL1))
    ↪ = if satisfies(get-event-time(EV),get-variable-name(PER1),
    ↪ get-time-constraint(PER1),VL1) and is-action-satisfied(get-
    ↪ id(PER1),get-action(PER1),EV,CAPL1)
then check-and-update-fulfillment( PER1,get-event-time(EV),
update-assets-with-event(get-id(PER1),get-action(PER1),EV,get-time-
    ↪ constraint(PER1),(
AL1 |
update-with-constraint(get-variable-name(PER1),get-event-time(EV),
    ↪ get-time-constraint(PER1),VL1) |
GS1 | CAPL1 | TL1
)
))
else check-and-update-fulfillment( PER1,get-event-time(EV), (AL1 |
    ↪ VL1 | GS1 | CAPL1 | TL1))
fi .
eq update-with-prohibition(PRO1,EV,(AL1 | VL1 | GS1 | CAPL1 | TL1))
    ↪  = if satisfies(get-event-time(EV),get-variable-name(PRO1),
    ↪ get-time-constraint(PRO1),VL1) and is-action-satisfied(get-
    ↪ id(PRO1),get-action(PRO1),EV,CAPL1)
then check-and-update-fulfillment( PRO1,get-event-time(EV),
update-assets-with-event(get-id(PRO1),get-action(PRO1),EV,get-time-
    ↪ constraint(PRO1),(
AL1 |
update-with-constraint(get-variable-name(PRO1),get-event-time(EV),
    ↪ get-time-constraint(PRO1),VL1) |
GS1 | CAPL1 | TL1
)
))
else check-and-update-fulfillment( PRO1,get-event-time(EV), (AL1 |
    ↪ VL1 | GS1 | CAPL1 | TL1))
fi .
ceq update-with-rule(RL1,EV,(AL1 | VL1 | GS1 | CAPL1 | TL1)) = (AL1
    ↪  | VL1 | GS1 | CAPL1 | TL1) if is-fulfilled(get-id(RL1),GS1)
    ↪  or is-breached(get-id(RL1),GS1) .
ceq update-with-rule(RL1,EV,(AL1 | VL1 | GS1 | CAPL1 | TL1)) = (AL1
    ↪  | VL1 | (get-id(RL1) fulfilled by get-event-time(EV) with
    ↪ time recorded in get-variable-name(RL1) ) GS1 | CAPL1 | TL1)
if is-fulfilled(get-condition-id(get-id(RL1)),GS1) and is-fulfilled
    ↪ (get-consequent-id(get-id(RL1)),GS1) .
ceq update-with-rule(RL1,EV,(AL1 | VL1 | GS1 | CAPL1 | TL1)) = (AL1
    ↪  | VL1 | (get-id(RL1) breached by get-event-time(EV) with
```

```
   ↪ time recorded in get-variable-name(RL1) ) GS1 | CAPL1 | TL1)
if is-breached(get-condition-id(get-id(RL1)),GS1) or is-breached(
   ↪ get-consequent-id(get-id(RL1)),GS1) .
ceq update-with-rule(CID1 ; if CND1 where time is VN1 then CNTS {
   ↪ CSL1},EV,(AL1 | VL1 | GS1 | CAPL1 | TL1)) = update-with-
   ↪ container(get-consequent-id(CID1) ; CNTS where time is get-
   ↪ consequent-vn(VN1) {CSL1}, EV,(AL1 | VL1 | GS1 | CAPL1 | TL1
   ↪ ))
if is-fulfilled(get-condition-id(CID1),GS1) .
eq update-with-rule(CID1 ; if (ACT1) (TC1) is satisfied where time
   ↪ is VN1 then CNTS {CSL1},EV,(AL1 | VL1 | GS1 | CAPL1 | TL1))
   ↪ =
if satisfies(get-event-time(EV),get-condition-vn(VN1),TC1,VL1) and
   ↪ is-action-satisfied(CID1,ACT1,EV,CAPL1)
then update-with-rule-special-case(CID1 ; if (ACT1) (TC1) is
   ↪ satisfied where time is VN1 then CNTS {CSL1},EV,check-and-
   ↪ update-fulfillment( get-condition-id(CID1) ,(ACT1) (TC1) is
   ↪ satisfied,get-condition-vn(VN1) ,get-event-time(EV),
update-assets-with-event(get-condition-id(CID1),ACT1,EV,TC1,(
AL1 |
update-with-constraint(get-condition-vn(VN1),get-event-time(EV),TC1
   ↪ ,VL1) |
GS1 | CAPL1 | TL1
))))
else check-and-update-fulfillment( get-consequent-id(CID1) ,(ACT1)
   ↪ (TC1) is satisfied,get-consequent-vn(VN1) ,get-event-time(EV
   ↪ ),check-and-update-fulfillment( get-condition-id(CID1) ,(
   ↪ ACT1) (TC1) is satisfied,get-condition-vn(VN1) ,get-event-
   ↪ time(EV),(AL1 | VL1 | GS1 | CAPL1 | TL1)))
fi .
eq update-with-rule(CID1 ; if (ACT1) (TC1) is not satisfied where
   ↪ time is VN1 then CNTS {CSL1},EV,(AL1 | VL1 | GS1 | CAPL1 |
   ↪ TL1)) =
if satisfies(get-event-time(EV),get-condition-vn(VN1),TC1,VL1) and
   ↪ is-action-satisfied(CID1,ACT1,EV,CAPL1)
then update-with-rule-special-case(CID1 ; if (ACT1) (TC1) is not
   ↪ satisfied where time is VN1 then CNTS {CSL1},EV,check-and-
   ↪ update-fulfillment( get-condition-id(CID1),(ACT1) (TC1) is
   ↪ satisfied ,get-condition-vn(VN1),get-event-time(EV),
update-assets-with-event(get-condition-id(CID1),ACT1,EV,TC1,(
AL1 |
update-with-constraint(get-condition-vn(VN1),get-event-time(EV),TC1
   ↪ ,VL1) |
GS1 | CAPL1 | TL1
))))
```

```
else check-and-update-fulfillment( get-consequent-id(CID1) ,(ACT1)
   ↪ (TC1) is satisfied,get-consequent-vn(VN1) ,get-event-time(EV
   ↪ ),check-and-update-fulfillment( get-condition-id(CID1) ,(
   ↪ ACT1) (TC1) is satisfied,get-condition-vn(VN1) ,get-event-
   ↪ time(EV),(AL1 | VL1 | GS1 | CAPL1 | TL1)))
fi .
eq update-with-rule-special-case(CID1 ; if (ACT1) (TC1) is
   ↪ satisfied where time is VN1 then CNTS {CSL1},EV,(AL1 | VL1 |
   ↪  GS1 | CAPL1 | TL1)) =
if is-fulfilled(get-condition-id(CID1),GS1)
then update-with-rule(CID1 ; if (ACT1) (TC1) is satisfied where
   ↪ time is VN1 then CNTS {CSL1},EV,(AL1 | VL1 | GS1 | CAPL1 |
   ↪ TL1))
else (AL1 | VL1 | GS1 | CAPL1 | TL1)
fi .
eq update-with-rule-special-case(CID1 ; if (ACT1) (TC1) is not
   ↪ satisfied where time is VN1 then CNTS {CSL1},EV,(AL1 | VL1 |
   ↪  GS1 | CAPL1 | TL1)) =
if is-fulfilled(get-condition-id(CID1),GS1)
then update-with-rule(CID1 ; if (ACT1) (TC1) is not satisfied where
   ↪  time is VN1 then CNTS {CSL1},EV,(AL1 | VL1 | GS1 | CAPL1 |
   ↪ TL1))
else (AL1 | VL1 | GS1 | CAPL1 | TL1)
fi .
ceq update-with-container(CID1 ; CNTS VD1 {CSL1},EV,(AL1 | VL1 |
   ↪ GS1 | CAPL1 | TL1)) = update-clauselist-state-given-event(
   ↪ CSL1,EV,(AL1 | VL1 | GS1 | CAPL1 | TL1)) if is-fulfilled(
   ↪ CID1,GS1) or is-breached(CID1,GS1) .
eq update-with-container(CID1 ; CNTS VD1 {CSL1},EV,CS1) =
update-with-container-special(CID1 ; CNTS VD1 {CSL1},EV,update-
   ↪ clauselist-state-given-event(CSL1,EV,CS1)) .
eq container-breached(any,noclauses,GS1) = true .
eq container-breached(all,noclauses,GS1) = false .
eq container-breached(all,CLAS1 CSL1,GS1) = if is-breached(get-id(
   ↪ CLAS1),GS1) then true else container-breached(all,CSL1,GS1)
   ↪ fi .
eq container-breached(any,CLAS1 CSL1,GS1) = if is-breached(get-id(
   ↪ CLAS1),GS1) then container-breached(any,CSL1,GS1) else false
   ↪  fi .
eq container-breached(CNTS,noclauses,GS1) = false .
eq container-breached(CNTS,CLAS1,GS1) = not(is-fulfilled(get-id(
   ↪ CLAS1),GS1)) .
eq container-breached(sequence,CLAS1 CLAS2 CSL1,GS1) =
if (is-breached(get-id(CLAS1),GS1) or is-breached(get-id(CLAS2),GS1
   ↪ )
```

```
or not(lt(get-recorded-time(find-state-of(get-id(CLAS1),GS1)),get-
   ↪ recorded-time(find-state-of(get-id(CLAS2),GS1)))))
then true else container-breached(sequence,CLAS2 CSL1,GS1) fi .
eq container-breached(parallel,CLAS1 CLAS2 CSL1,GS1) =
if is-fulfilled(get-id(CLAS1),GS1) or is-fulfilled(get-id(CLAS2),
   ↪ GS1)
or not(get-recorded-time(find-state-of(get-id(CLAS1),GS1)) == get-
   ↪ recorded-time(find-state-of(get-id(CLAS2),GS1)))
then true
else container-breached(parallel,CLAS2 CSL1,GS1) fi .
eq container-satisfied(all,noclauses,GS1) = true .
eq container-satisfied(any,noclauses,GS1) = false .
eq container-satisfied(all,CLAS1 CSL1,GS1) = if not(is-fulfilled(
   ↪ get-id(CLAS1),GS1)) then false else container-satisfied(all,
   ↪ CSL1,GS1) fi .
eq container-satisfied(any,CLAS1 CSL1,GS1) = if is-fulfilled(get-id
   ↪ (CLAS1),GS1) then true else container-satisfied(any,CSL1,GS1
   ↪ ) fi .
eq container-satisfied(CNTS,noclauses,GS1) = true .
eq container-satisfied(CNTS,CLAS1,GS1) = is-fulfilled(get-id(CLAS1)
   ↪ ,GS1) .
eq container-satisfied(sequence,CLAS1 CLAS2 CSL1,GS1) =
is-fulfilled(get-id(CLAS1),GS1)
and is-fulfilled(get-id(CLAS2),GS1)
and lt(get-recorded-time(find-state-of(get-id(CLAS1),GS1)),get-
   ↪ recorded-time(find-state-of(get-id(CLAS2),GS1)))
and container-satisfied(sequence,CLAS2 CSL1,GS1) .
eq container-satisfied(parallel,CLAS1 CLAS2 CSL1,GS1) =
is-fulfilled(get-id(CLAS1),GS1)
and is-fulfilled(get-id(CLAS2),GS1)
and get-recorded-time(find-state-of(get-id(CLAS1),GS1)) == get-
   ↪ recorded-time(find-state-of(get-id(CLAS2),GS1))
and container-satisfied(parallel,CLAS2 CSL1,GS1) .
eq update-with-container-special(CID1 ; CNTS where time is VN1 {
   ↪ CSL1},EV,(AL1 | VL1 | GS1 | CAPL1 | TL1)) =
if container-satisfied(CNTS,CSL1,GS1)
then check-and-update-fulfillment(CID1 ; CNTS where time is VN1 {
   ↪ CSL1},get-event-time(EV),
(
AL1 |
update-with-constraint(VN1,get-event-time(EV),get-time-constraint(
   ↪ CID1 ; CNTS where time is VN1 {CSL1}),VL1) |
GS1 | CAPL1 | TL1
))
```

```
else check-and-update-fulfillment(CID1 ; CNTS where time is VN1 {
   ↪ CSL1},get-event-time(EV),
(AL1 | VL1 | GS1 | CAPL1 | TL1))
fi .
ceq check-and-update-fulfillment(CID1,CND1,VN1,T1, (AL1 | VL1 | GS1
   ↪  | CAPL1 | TL1)) = (AL1 | VL1 | (CID1 fulfilled by T1 with
   ↪ time recorded in VN1) GS1 | CAPL1 | TL1) if check-fulfilled(
   ↪ CND1,VN1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) .
ceq check-and-update-fulfillment(CID1,CND1,VN1,T1, (AL1 | VL1 | GS1
   ↪  | CAPL1 | TL1)) = (AL1 | VL1 | (CID1 breached by T1 with
   ↪ time recorded in VN1) GS1 | CAPL1 | TL1) if check-breached(
   ↪ CND1,VN1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) .
eq check-and-update-fulfillment(CID1,CND1,VN1,T1, (AL1 | VL1 | GS1
   ↪ | CAPL1 | TL1)) = (AL1 | VL1 | GS1 | CAPL1 | TL1) .
ceq check-and-update-fulfillment(PER1,T1,(AL1 | VL1 | GS1 | CAPL1 |
   ↪  TL1)) = (AL1 | VL1 | (get-id(PER1) exercised by T1 with
   ↪ time recorded in get-variable-name(PER1)) GS1 | CAPL1 | TL1)
   ↪  if check-fulfilled(PER1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1))
   ↪  .
ceq check-and-update-fulfillment(PER1,T1,(AL1 | VL1 | GS1 | CAPL1 |
   ↪  TL1)) = (AL1 | VL1 | (get-id(PER1) not exercised by T1 with
   ↪  time recorded in get-variable-name(PER1)) GS1 | CAPL1 | TL1
   ↪ ) if check-breached(PER1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1))
   ↪  .
ceq check-and-update-fulfillment(CNT1,T1,(AL1 | VL1 | GS1 | CAPL1 |
   ↪  TL1)) = (AL1 | VL1 | (get-id(CNT1) fulfilled by T1 with
   ↪ time recorded in get-variable-name(CNT1)) GS1 | CAPL1 | TL1)
   ↪  if check-fulfilled(CNT1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1))
   ↪  .
ceq check-and-update-fulfillment(CNT1,T1,(AL1 | VL1 | GS1 | CAPL1 |
   ↪  TL1)) = (AL1 | VL1 | (get-id(CNT1) breached by T1 with time
   ↪  recorded in get-variable-name(CNT1)) GS1 | CAPL1 | TL1) if
   ↪ check-breached(CNT1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) .
ceq check-and-update-fulfillment(O1,T1,(AL1 | VL1 | GS1 | CAPL1 |
   ↪ TL1)) = (AL1 | VL1 | (get-id(O1) fulfilled by T1 with time
   ↪ recorded in get-variable-name(O1)) GS1 | CAPL1 | TL1) if
   ↪ check-fulfilled(O1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) .
ceq check-and-update-fulfillment(O1,T1,(AL1 | VL1 | GS1 | CAPL1 |
   ↪ TL1)) = (AL1 | VL1 | (get-id(O1) breached by T1 with time
   ↪ recorded in get-variable-name(O1)) GS1 | CAPL1 | TL1) if
   ↪ check-breached(O1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) .
ceq check-and-update-fulfillment(PRO1,T1,(AL1 | VL1 | GS1 | CAPL1 |
   ↪  TL1)) = (AL1 | VL1 | (get-id(PRO1) fulfilled by T1 with
   ↪ time recorded in get-variable-name(PRO1)) GS1 | CAPL1 | TL1)
   ↪  if check-fulfilled(PRO1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1))
```

```
   ↪  .
ceq check-and-update-fulfillment(PRO1,T1,(AL1 | VL1 | GS1 | CAPL1 |
   ↪  TL1)) = (AL1 | VL1 | (get-id(PRO1) breached by T1 with time
   ↪  recorded in get-variable-name(PRO1)) GS1 | CAPL1 | TL1) if
   ↪ check-breached(PRO1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) .
eq check-and-update-fulfillment(CLAS1,T1,(AL1 | VL1 | GS1 | CAPL1 |
   ↪  TL1)) = (AL1 | VL1 | GS1 | CAPL1 | TL1) .
eq is-fulfilled(CID1,GS1) = get-clause-state-type(find-state-of(
   ↪ CID1,GS1)) == fulfillment-state .
eq is-breached(CID1,GS1) = get-clause-state-type(find-state-of(CID1
   ↪ ,GS1)) == breach-state .
eq check-fulfilled((ACT1) (TC1) is satisfied,VN1,T1,(AL1 | VL1 |
   ↪ GS1 | CAPL1 | TL1)) = is-variable-count-match(VN1,TC1,VL1)
   ↪ and is-tc-dependency-fulfilled(T1,TC1,(AL1 | VL1 | GS1 |
   ↪ CAPL1 | TL1)) .
eq check-breached((ACT1) (TC1) is satisfied,VN1,T1,(AL1 | VL1 | GS1
   ↪  | CAPL1 | TL1)) = exceeds(T1,TC1,VL1) and (not check-
   ↪ fulfilled((ACT1) (TC1) is satisfied,VN1,T1,(AL1 | VL1 | GS1
   ↪ | CAPL1 | TL1) )) .
eq check-fulfilled((ACT1) (TC1) is not satisfied,VN1,T1,(AL1 | VL1
   ↪ | GS1 | CAPL1 | TL1)) = not(is-variable-count-match(VN1,TC1,
   ↪ VL1) and is-tc-dependency-fulfilled(T1,TC1,(AL1 | VL1 | GS1
   ↪ | CAPL1 | TL1))) .
eq check-breached((ACT1) (TC1) is not satisfied,VN1,T1,(AL1 | VL1 |
   ↪  GS1 | CAPL1 | TL1)) = not (exceeds(T1,TC1,VL1) and not
   ↪ check-fulfilled((ACT1) (TC1) is not satisfied,VN1,T1,(AL1 |
   ↪ VL1 | GS1 | CAPL1 | TL1))) .
eq check-fulfilled(CID1 ; CNTS where time is VN1 {CSL1},T1,(AL1 |
   ↪ VL1 | GS1 | CAPL1 | TL1)) = is-variable-count-match(VN1,at
   ↪ any time,VL1) and is-tc-dependency-fulfilled(T1,at any time
   ↪ ,(AL1 | VL1 | GS1 | CAPL1 | TL1)) .
eq check-breached(CID1 ; CNTS where time is VN1{CSL1},T1,(AL1 | VL1
   ↪  | GS1 | CAPL1 | TL1)) = container-breached(CNTS,CSL1,GS1) .
eq check-fulfilled(O1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) = is-
   ↪ variable-count-match(get-variable-name(O1),get-time-
   ↪ constraint(O1),VL1)
and is-tc-dependency-fulfilled(T1,get-time-constraint(O1),(AL1 |
   ↪ VL1 | GS1 | CAPL1 | TL1)) .
eq check-breached(O1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) = exceeds(
   ↪ T1,get-time-constraint(O1),VL1) and not check-fulfilled(O1,
   ↪ T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) .
eq check-fulfilled(PER1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) = is-
   ↪ variable-count-match(get-variable-name(PER1),get-time-
   ↪ constraint(PER1),VL1)
```

```
and is-tc-dependency-fulfilled(T1,get-time-constraint(PER1),(AL1 |
    ↪ VL1 | GS1 | CAPL1 | TL1)) .
eq check-breached(PER1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) =
    ↪ exceeds(T1,get-time-constraint(PER1),VL1) and not( check-
    ↪ fulfilled(PER1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) ) .
eq check-fulfilled(PRO1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) =
    ↪ exceeds(T1,get-time-constraint(PRO1),VL1) and not check-
    ↪ breached(PRO1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) and is-tc-
    ↪ dependency-fulfilled(T1,get-time-constraint(PRO1),(AL1 | VL1
    ↪  | GS1 | CAPL1 | TL1)) .
eq check-breached(PRO1,T1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) = is-
    ↪ variable-count-match(get-variable-name(PRO1),get-time-
    ↪ constraint(PRO1),VL1)
and is-tc-dependency-fulfilled(T1,get-time-constraint(PRO1),(AL1 |
    ↪ VL1 | GS1 | CAPL1 | TL1)) .
eq is-variable-count-match(VN1,at any time repeated,VL1) = count-
    ↪ variables(get-variable-list(get-variable-group(VN1,VL1)))
>= count-increments(at any time repeated,VL1) .
eq is-variable-count-match(VN1,at all times,VL1) = count-variables(
    ↪ get-variable-list(get-variable-group(VN1,VL1)))
>= count-increments(at all times,VL1) .
eq is-variable-count-match(VN1,TC1,VL1) = (not(get-single-variable-
    ↪ time-value(VN1,VL1) == undefined-time))
or (count-variables(get-variable-list(get-variable-group(VN1,VL1)))
    ↪  == (count-increments(TC1,VL1))) .
eq is-tc-dependency-fulfilled(T1,at each VN1,(AL1 | VL1 | GS1 |
    ↪ CAPL1 | TL1)) = not (find-state-by-vn(VN1,GS1) ==
    ↪ noclausestate) .
eq is-tc-dependency-fulfilled(T1,within T1 of each VN1,(AL1 | VL1 |
    ↪  GS1 | CAPL1 | TL1)) = not (find-state-by-vn(VN1,GS1) ==
    ↪ noclausestate) .
eq is-tc-dependency-fulfilled(T1,at any time repeated,(AL1 | VL1 |
    ↪ GS1 | CAPL1 | TL1)) = exceeds(T1,at any time repeated,VL1) .
eq is-tc-dependency-fulfilled(T1,TC1,(AL1 | VL1 | GS1 | CAPL1 | TL1
    ↪ )) = variables-are-defined(TC1,VL1) .
eq is-action-satisfied(CID1,TRA1 by PID1 for PID2, TRA2 by PID3 for
    ↪  PID4 @ T1 in fulfillment of CID2,CAPL1) = CID1 == CID2 and
    ↪ PID1 == PID3 and PID2 == PID4 and TRA1 == TRA2 .
eq is-action-satisfied(CID1,TRA1 by PID1 for PID2, DACT1 @ T1,CAPL1
    ↪ ) = false .
eq is-action-satisfied(CID1,DACT1, TRA1 by PID1 for PID2 @ T1 in
    ↪ fulfillment of CID2,CAPL1) = false .
eq is-action-satisfied(CID1,DACT1, DACT2 @ T1,CAPL1) = DACT1 ==
    ↪ DACT2 .
```

```
eq update-asset-clause-pairs(CID1,transfer N1 ASC1, transfer N2
   ↪ ASC2 by PID1 for PID2 , TC1, VL1,nopairs) = CID1 ; ( (N2 * (
   ↪ count-increments(TC1,VL1)) - N1 )) .
eq update-asset-clause-pairs(CID1,transfer N1 ASC1 , ACT1, TC1, VL1
   ↪ ,(CID2 ; I1) CAPL1) =
if CID1 == CID2
then (CID2 ; (I1 - N1) ) CAPL1
else (CID2 ; I1) update-asset-clause-pairs(CID1,transfer N1 ASC1 ,
   ↪ ACT1, TC1,VL1, CAPL1)
fi .
eq update-asset-clause-pairs(CID1,TRA1, ACT1 , TC1, VL1,CAPL1) =
   ↪ CAPL1 .
ceq update-assets-with-event(CID1,ACT1,TRA1 by PID1 for PID2 @ T1
   ↪ in fulfillment of CID2,TC1,(AL1 | VL1 | GS1 | CAPL1 | TL1))
   ↪ = (AL1 | VL1 | GS1 | update-asset-clause-pairs(CID1,TRA1,
   ↪ ACT1,TC1,VL1,CAPL1) | TL1) if CID1 == CID2 .
eq update-assets-with-event(CID1,ACT1,TRA1 by PID1 for PID2 @ T1 in
   ↪  fulfillment of CID2,TC1,(AL1 | VL1 | GS1 | CAPL1 | TL1)) =
   ↪ (AL1 | VL1 | GS1 | update-asset-clause-pairs(CID1,TRA1,ACT1,
   ↪ TC1,VL1,CAPL1) | TL1) .
eq update-assets-with-event(CID1,ACT1,ACT2 @ T1,TC1,(AL1 | VL1 |
   ↪ GS1 | CAPL1 | TL1)) = (AL1 | VL1 | GS1 | CAPL1 | TL1) .
eq update-transaction-list(TRA1 by PID1 for PID2 @ T1 in
   ↪ fulfillment of CID1 , TL1) = (transaction ; (TRA1 by PID1
   ↪ for PID2 @ T1 in fulfillment of CID1)) TL1 .
eq update-transaction-list(EV ,TL1) = TL1 .
endfm
```

APPENDIX B

BASIC TESTS

B.1 Actions Module

```
load ../../specification/SLANC.maude .
parse act .
parse nilaction .
parse transfer 5 SAR .
parse transfer 11 USD .
parse do "xxxxxxs" .
parse do "expediate" .
parse do "sell" by 'elkoko for 'elmazeene .
parse transfer 100 USD by 'mazene for 'lacoco .
quit .
```

B.2 Agreements Module

```
load ../../specification/SLANC.maude .
parse "xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where
   ↪ time is "agreement time" .
parse "xv" ; 'a is permitted to act for 'b at (15 , 5 , 5) where
   ↪ time is "agreement time" .
parse "xv" ; 'a is prohibited from act for 'b at (15 , 5 , 5) where
   ↪  time is "agreement time" .
parse "xv" ; 'a is prohibited from do "AAAA" for 'b at (15 , 5 , 5)
   ↪  where time is "agreement time" .
parse override ("xv" ; 'a is prohibited from do "AAAA" for 'b at
   ↪ (15 , 5 , 5) where time is "agreement time") .
```

```
quit .
```

B.3 Assets Module

```
load ../../specification/SLANC.maude .
parse noassets .
parse { name: "Saudi Ryials", code: SAR , supply: 500000, owner: '
   ↪ ElKoko } .
parse { name: "Saudi Ryials", code: SAR , supply: 500000, owner: '
   ↪ ElKoko } { name: "Saudi Ryials", code: SAR , supply: 500000,
   ↪  owner: 'ElKoko } .
parse { name: "Saudi Ryials", code: SAR , supply: 500000, owner: '
   ↪ ElKoko } { name: "Saudi Ryials", code: SAR , supply: 500000,
   ↪  owner: 'ElKoko } { name: "Saudi Ryials", code: SAR , supply
   ↪ : 500000, owner: 'ElKoko } .
red update-assets-with-event( "AA",transfer 1001 SAR by 'E for 'A,
   ↪ transfer 0 SAR by 'E for 'A @ (1 , 2 , 1) in fulfillment of
   ↪ "AA", (every (1 , 0 , 0) starting at (1 , 1 , 1) and ending
   ↪ at (5 , 1 , 1)) ,({ name: "Saudi Ryials", code: SAR , supply
   ↪ : 500000, owner: 'E } { name: "Saudi Ryials", code: SAR ,
   ↪ supply: 500000, owner: 'A } | novariables | noglobalstate |
   ↪ nopairs | notransactions)) .
quit .
```

B.4 Conditions Module

```
load ../../specification/SLANC.maude .
parse nilaction at (5 , 5 , 5) is satisfied .
parse nilaction within (5 , 5 , 5) of "good time" is satisfied .
parse do "xyz" by 'elmo for 'ww within 5 , 5 , 5 of "good time" is
   ↪ satisfied .
parse "xyz" by 'elmo for 'ww within (5 , 5 , 5) of "good time" is
   ↪ satisfied .
parse transfer 3 SAR by 'abc for 'ee within (5 , 5 , 5) of "good
   ↪ time" is satisfied .
parse transfer 5 USD by 'nn for 'bb at (5 , 5 , 5) is satisfied .
parse (transfer 5 USD by 'nn for 'bb) every (5 , 5 , 5) starting at
   ↪  (5 , 5 , 5) and ending at (100 , 5 , 5) is satisfied .
quit .
```

B.5 Time Constraints Module

```
load ../../specification/SLANC.maude .
parse at (5 , 5 , 5) .
parse at "koko" .
parse before "anker" .
parse after (15 , 5 , 5) .
parse between (15 , 5 , 5) and (15 , 5 , 5) .
parse between (15 , 5 , 5) and "bath time" + (15 , 5 , 5) .
parse between (15 , 5 , 5) and "sho sho" .
parse within (15 , 5 , 5) of "caster" .
parse every (0 , 0 , 1) starting at "A" and ending at "B" .
parse every (0 , 0 , 1) starting at (0 , 0 , 1) and ending at "B" .
parse every (0 , 0 , 1) starting at "A" and ending at (0 , 0 , 1) .
parse every (0 , 0 , 1) starting at (0 , 0 , 1) and ending at (0 ,
   ↪ 0 , 1) .
parse at all times .
parse at each "koko" .
parse within (15 , 5 , 5) of each "caster" .
red satisfies((15 , 5 , 5) ,"XB",at (15 , 5 , 5),novariables) .
red satisfies((15 , 5 , 5) ,"XB",at ("A" + (0 , 0 , 1)),("A" is (15
   ↪  , 5 , 4)) ) .
red satisfies((15 , 5 , 5) ,"XB",at (15 , 5 , 6),novariables) .
red satisfies((15 , 5 , 5) ,"XB",after (15 , 5 , 5),novariables) .
red satisfies((15 , 5 , 5) ,"XB",after ("A" + (0 , 0 , 1)),("A" is
   ↪ (15 , 5 , 3)) ) .
red satisfies((15 , 5 , 5) ,"XB",after (15 , 5 , 4),novariables) .
red satisfies((15 , 5 , 5) ,"XB",before (15 , 5 , 5),novariables) .
red satisfies((15 , 5 , 5) ,"XB",before ("A" + (0 , 0 , 1)),("A" is
   ↪  (15 , 5 , 5)) ) .
red satisfies((15 , 5 , 5) ,"XB",before (15 , 5 , 6),novariables) .
red satisfies((15 , 5 , 4) ,"XB",between (15 , 5 , 5) and (15 , 5 ,
   ↪  7) ,novariables) .
red satisfies((15 , 5 , 8) ,"XB",between (15 , 5 , 5) and (15 , 5 ,
   ↪  7) ,novariables) .
red satisfies((15 , 5 , 5) ,"XB",between (15 , 5 , 5) and (15 , 5 ,
   ↪  7) ,novariables) .
red satisfies((15 , 5 , 7) ,"XB",between (15 , 5 , 5) and (15 , 5 ,
   ↪  7) ,novariables) .
red satisfies((15 , 5 , 6) ,"XB",between (15 , 5 , 5) and (15 , 5 ,
   ↪  7) ,novariables) .
red satisfies((15 , 5 , 5) ,"XB",between ("A" + (0 , 0 , 1)) and ("
   ↪ A" + (0 , 0 , 5)) ,("A" is (15 , 5 , 3)) ) .
red satisfies((15 , 5 , 3) ,"XB",between ("A" + (0 , 0 , 1)) and ("
```

```
   ↪ A" + (0 , 0 , 5)) ,("A" is (15 , 5 , 3)) ) .
red satisfies((15 , 5 , 4) ,"XB",within (0 , 0 , 5) of (15 , 5 , 5)
   ↪  ,novariables) .
red satisfies((15 , 5 , 5) ,"XB",within (0 , 0 , 5) of ("A" + (0 ,
   ↪ 0 , 1)) ,("A" is (15 , 5 , 3)) ) .
red satisfies((15 , 5 , 4) ,"XB",within (0 , 0 , 5) of ("A" + (0 ,
   ↪ 0 , 1)) ,("A" is (15 , 5 , 3)) ) .
red satisfies((15 , 5 , 5) ,"XB",within (0 , 0 , 5) of (15 , 5 , 5)
   ↪  ,novariables) .
red satisfies((15 , 5 , 6) ,"XB",within (0 , 0 , 5) of (15 , 5 , 5)
   ↪  ,novariables) .
red satisfies((15 , 5 , 7) ,"XB", within (0 , 0 , 5) of (15 , 5 ,
   ↪ 5) ,novariables) .
red satisfies((15 , 5 , 8) ,"XB", within (0 , 0 , 5) of (15 , 5 ,
   ↪ 5) ,novariables) .
red satisfies((15 , 5 , 9) ,"XB", within (0 , 0 , 5) of (15 , 5 ,
   ↪ 5) ,novariables) .
red satisfies((15 , 5 , 10) ,"XB",within (0 , 0 , 5) of (15 , 5 ,
   ↪ 5) ,novariables) .
red satisfies((15 , 5 , 11) ,"XB", within (0 , 0 , 5) of (15 , 5 ,
   ↪ 5) ,novariables) .
red satisfies((15 , 5 , 6) ,"XB",every (0 , 0 , 1) starting at (15
   ↪ , 5 , 5) and ending at (15 , 5 , 9) ,novariables) .
red satisfies((15 , 5 , 6) ,"XB",every (0 , 0 , 1) starting at (15
   ↪ , 5 , 5) and ending at (15 , 5 , 9) ,"XB" group is { "XB" #
   ↪ 1 is (15 , 5 , 5) }) .
red satisfies((15 , 5 , 5) ,"XB",every (0 , 0 , 1) starting at (15
   ↪ , 5 , 5) and ending at (15 , 5 , 9) ,"XB" group is { "XB" #
   ↪ 1 is (15 , 5 , 6) }) .
red satisfies((15 , 5 , 5) ,"XB",every (0 , 0 , 1) starting at ("A"
   ↪  + (0 , 0 , 1)) and ending at (15 , 5 , 9) ,("XB" group is {
   ↪  "XB" # 1 is (15 , 5 , 6)}) ("A" is (15 , 5 , 4)) ) .
red satisfies((15 , 5 , 5) ,"XB", at each "A" ,("A" group is { "A"
   ↪ # 1 is (15 , 5 , 5)}) ("XB" is (15 , 5 , 4)) ) .
red satisfies((15 , 5 , 6) ,"XB", at each ("A" + (0 , 0 , 1)) ,("A"
   ↪  group is { "A" # 1 is (15 , 5 , 5)}) ("XB" is (15 , 5 , 4))
   ↪  ) .
red satisfies((15 , 5 , 9) ,"XB", within (0 , 0 , 5) of each "A"
   ↪ ,("A" group is { "A" # 1 is (15 , 5 , 5)}) ("XB" is (15 , 5
   ↪ , 4))) .
red satisfies((15 , 5 , 11) ,"XB", within (0 , 0 , 5) of each "A"
   ↪ ,("A" group is { "A" # 1 is (15 , 5 , 5)}) ("XB" is (15 , 5
   ↪ , 4))) .
red make-new-group-variable("A",(15 , 5 , 11),("A" group is { "A" #
   ↪  1 is (15 , 5 , 5)}) ("XB" is (15 , 5 , 4))) .
```

```
quit .
```

B.6 Containers Module

```
load ../../specification/SLANC.maude .
parse "a" ; all (where time is "first time") { noclauses } .
parse "a" ; all (where time is "first time") { noclauses } .
parse "a" ; all (where time is "first time") { noclauses } .
parse "d" ; any (where time is "first time") { noclauses } .
parse "c" ; all (where time is "first time") {
("xv" ; 'a is permitted to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
(override ("xv" ; 'a is permitted to act for 'b at (15 , 5 , 5)
   ↪ where time is "agreement time"))
("xv" ; 'a is prohibited from act for 'b at (15 , 5 , 5) where time
   ↪  is "agreement time")
("xv" ; 'a is prohibited from act for 'b at (15 , 5 , 5) where time
   ↪  is "agreement time") } .
parse "b" ; any (where time is "first time") {
("xv" ; 'a is prohibited from act for 'b at (15 , 5 , 5) where time
   ↪  is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time") } .
quit .
```

B.7 Contract Extension Semantics Module

```
load ../../specification/SLANC.maude .
parse extend-all(Contract,Contract Contract Contract ) .
parse extend-single(Contract,Contract) .
parse extend-preamble(Preamble,Preamble) .
parse extend-body(Body,Body) .
parse resolve-overrides(noclauses,noclauses) .
parse resolve-override(nilclause,noclauses) .
red get-id("1" ; 'b is obligated to act for 'a at (15 , 5 , 5)
   ↪ where time is "agreement 33 time") .
red resolve-override(override ("1" ; 'b is obligated to act for 'a
   ↪ at (15 , 5 , 5) where time is "agreement 33 time"),("1" ; 'a
```

```
   ↪  is obligated to act for 'b at (15 , 5 , 5) where time is "
   ↪ agreement time")) .
red resolve-override(override ("1" ; 'b is obligated to act for 'a
   ↪ at (15 , 5 , 5) where time is "agreement 33 time"),
override ("1" ; if nilaction at (15 , 5 , 5) is satisfied where
   ↪ time is "good time" then
all {("xv" ; 'a is permitted to act for 'b after (5 , 0 , 0) where
   ↪ time is "agreement time")
("xv" ; 'a is permitted to act for 'b between (1 , 0 , 0) and "play
   ↪  time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
add (15 , 5 , 5) to "var 1" }) ) .
red resolve-override(override ("1" ; 'b is obligated to act for 'a
   ↪ at (15 , 5 , 5) where time is "agreement 33 time"),
"2" ; if nilaction at (15 , 5 , 5) is satisfied where time is "good
   ↪  time" then
all {("xv" ; 'a is permitted to act for 'b after (5 , 0 , 0) where
   ↪ time is "agreement time")
("xv" ; 'a is permitted to act for 'b between (1 , 0 , 0) and "play
   ↪  time" where time is "agreement time")
("1" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
add (15 , 5 , 5) to "var 1" } ) .
red resolve-override(override ("1" ; 'b is obligated to act for 'a
   ↪ at (15 , 5 , 5) where time is "agreement 33 time"),
"2" ; if nilaction at (15 , 5 , 5) is satisfied where time is "good
   ↪  time" then
all {("xv" ; 'a is permitted to act for 'b after (15 , 5 , 5) where
   ↪  time is "agreement time")
("b" ; any (where time is "first time") {
("xv" ; 'a is prohibited from act for 'b at (15 , 5 , 5) where time
   ↪  is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("1" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time") })
("cc" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
```

```
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
add (15 , 5 , 5) to "var 1" } ) .
red resolve-overrides(override ("1" ; 'b is obligated to act for 'a
   ↪  at (15 , 5 , 5) where time is "agreement 33 time"),("1" ; '
   ↪ a is obligated to act for 'b at (15 , 5 , 5) where time is "
   ↪ agreement time")
("1" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")) .
red resolve-overrides(override ("1" ; 'b is obligated to act for 'a
   ↪  at (15 , 5 , 5) where time is "agreement 33 time"),
("1" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")
("2" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")) .
red resolve-overrides(override ("1" ; 'b is obligated to act for 'a
   ↪  at (1 , 1 , 1) where time is "agreement 33 time"),
("xx" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("2" ; if nilaction at (15 , 5 , 5) is satisfied where time is "
   ↪ good time" then
all {("xv" ; 'a is permitted to act for 'b after (5 , 0 , 0) where
   ↪ time is "agreement time")
("b" ; any (where time is "first time") {
("xv" ; 'a is prohibited from act for 'b at (15 , 5 , 5) where time
   ↪  is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("1" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time") })
("cc" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
add (5 , 5 , 5) to "var 1" })) .
red resolve-overrides(override ("b" ; any (where time is "first
   ↪ time") { ("xv" ; 'a is prohibited from act for 'b at (15 , 5
   ↪  , 5) where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("1" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
```

```
   ↪ "agreement time") }),("b" ; 'a is obligated to act for 'b at
   ↪  (15 , 5 , 5) where time is "agreement time")) .
red resolve-overrides(override ("qwer" ; any (where time is "first
   ↪ time") { ("xv" ; 'a is prohibited from act for 'b at (15 , 5
   ↪  , 5) where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time") }),("b" ; 'a is obligated to act for 'b
   ↪ at (15 , 5 , 5) where time is "agreement time")
("2" ; if nilaction at (15 , 5 , 5) is satisfied where time is "
   ↪ good time" then
all {("xv" ; 'a is permitted to act for 'b after (15 , 5 , 5) where
   ↪  time is "agreement time")
("b" ; any (where time is "first time") {
("xv" ; 'a is prohibited from act for 'b at (15 , 5 , 5) where time
   ↪  is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("qwer" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time
   ↪ is "agreement time")
("1" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time") })
("cc" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
add (15 , 5 , 5) to "var 1" })) .
red extend-all(Contract,Contract Contract Contract) .
red extend-all(Contract,Contract Contract Contract{Preamble Body})
   ↪ .
red extend-all(Contract,Contract Contract Contract{
Preamble{
name: "My Contract1"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}]
extends: none
} Body { ("1" ; 'a is obligated to act for 'b at (15 , 5 , 5) where
```

```
    ↪  time is "agreement time")
("2" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
    ↪ "agreement time")
("3" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
    ↪ "agreement time")}}) .
red extend-all(Contract,Contract{Preamble{
name: "My Contract2"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
    ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
    ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
    ↪ noaddress , assets: [noassets]}]
extends: none
} Body { (
"1" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is "
    ↪ agreement time")
("2" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
    ↪ "agreement time")
("3" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
    ↪ "agreement time")}} Contract{Preamble{
name: "My Contract3"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
    ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
    ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
    ↪ noaddress , assets: [noassets]}]
extends: none
} Body {
("4" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
    ↪ "agreement time")
("5" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
    ↪ "agreement time")
("6" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
    ↪ "agreement time")}} Contract{Preamble{
name: "My Contract4"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
    ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
    ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
```

```
   ↪ noaddress , assets: [noassets]}]
extends: none
} Body { ("7" ; 'a is obligated to act for 'b at (15 , 5 , 5) where
   ↪  time is "agreement time")
("8" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")
("9" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")}}) .
red extend-all(Contract{Preamble{
name: "My NEW Contract"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}]
extends: "My Contract1"
} Body {
(override ("2" ; 'xxxx is obligated to act for 'yyyy at (15 , 5 ,
   ↪ 5) where time is "1(15 , 5 , 5) time"))
(override ("8" ; 'zzz is obligated to act for 'ccc at (15 , 5 , 5)
   ↪ where time is "the time"))
}},Contract{Preamble{
name: "My Contract1"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}]
extends: none
} Body { ("1" ; 'a is obligated to act for 'b at (15 , 5 , 5) where
   ↪  time is "agreement time")
("2" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")
("3" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")}} Contract{Preamble{
name: "My Contract2"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}
```

```
{ id: 'abs , name: "abssss" , description: "deco" , address:
    ↪ noaddress , assets: [noassets]}]
extends: none
} Body { ("4" ; 'a is obligated to act for 'b at (15 , 5 , 5) where
    ↪  time is "agreement time")
("5" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
    ↪ "agreement time")
("6" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
    ↪ "agreement time")}} Contract{Preamble{
name: "My Contract3"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
    ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
    ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
    ↪ noaddress , assets: [noassets]}]
extends: none
} Body { ("7" ; 'a is obligated to act for 'b at (15 , 5 , 5) where
    ↪  time is "agreement time")
("8" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
    ↪ "agreement time")
("9" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
    ↪ "agreement time")}}) .
red resolve-extensions(Contract{Preamble{
name: "My NEW Contract"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
    ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
    ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
    ↪ noaddress , assets: [noassets]}]
extends: "My Contract1"
} Body {
(override ("2" ; 'xxxx is obligated to act for 'yyyy at (15 , 5 ,
    ↪ 5) where time is "1(15 , 5 , 5) time"))
(override ("8" ; 'zzz is obligated to act for 'ccc at (15 , 5 , 5)
    ↪ where time is "the time"))
}},Contract{Preamble{
name: "My Contract1"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
    ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
```

```
   ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}]
extends: none
} Body { ("1" ; 'a is obligated to act for 'b at (15 , 5 , 5) where
   ↪  time is "agreement time")
("2" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")
("3" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")}} Contract{Preamble{
name: "My Contract2"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}]
extends: none
} Body { ("4" ; 'a is obligated to act for 'b at (15 , 5 , 5) where
   ↪  time is "agreement time")
("5" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")
("6" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")}} Contract{Preamble{
name: "My Contract3"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}]
extends: none
} Body { ("7" ; 'a is obligated to act for 'b at (15 , 5 , 5) where
   ↪  time is "agreement time")
("8" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")
("9" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪ "agreement time")}}) .
quit .
```

B.8 Contracts Module

```
load ../../specification/SLANC.maude .
parse Contract .
parse Body .
parse Body {noclauses} .
parse Contract{
Preamble{
name: "My Contract"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}]
extends: none}
Body{
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; if (nilaction at (15 , 5 , 5) is satisfied) (where time is
   ↪ "good time") then all {("xv" ; 'a is obligated to act for 'b
   ↪  after (15 , 5 , 5) where time is "agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
add (15 , 5 , 5) to "var 1" })
("xv" ; if nilaction every (15 , 5 , 5) starting at (5 , 5 , 5) and
   ↪  ending at (15 , 5 , 5) is satisfied where time is "good
   ↪ time" then all {noclauses})
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (5 , 5 , 5) and ending at (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then any {noclauses})
("xv" ; if do "string" by 'eko for 'ddn before (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then any {("xv" ; 'a is
   ↪ obligated to act for 'b after (15 , 5 , 5) where time is "
   ↪ agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
```

```
(subtract (15 , 5 , 5) from "var 2")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (5 , 5 , 5) and ending at (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then
any {noclauses})})
("xv" ; if (nilaction at (15 , 5 , 5) is not satisfied) (where time
   ↪  is "good time") then all {("xv" ; 'a is obligated to act
   ↪ for 'b after (15 , 5 , 5) where time is "agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")})
("xv" ; if nilaction every (15 , 5 , 5) starting at (5 , 5 , 5) and
   ↪  ending at (15 , 5 , 5) is not satisfied where time is "good
   ↪  time" then all {noclauses})
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (5 , 5 , 5) and ending at (15 , 5 , 5) is not
   ↪ satisfied where time is "good time" then any {noclauses})
("xv" ; if do "string" by 'eko for 'ddn before (15 , 5 , 5) is not
   ↪ satisfied where time is "good time" then any {("xv" ; 'a is
   ↪ obligated to act for 'b after (15 , 5 , 5) where time is "
   ↪ agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (5 , 5 , 5) and ending at (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then
any {noclauses})})}} .
red get-contract-name(Contract{
Preamble{
name: "My Contract"
start time: (15 , 5 , 5)
end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: "the address" , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address: "the
   ↪ address" , assets: [noassets]}
```

```
{ id: 'abs , name: "abssss" , description: "deco" , address: "the
   ↪ address" , assets: [noassets]}]
extends: none}
Body{
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; if nilaction at (15 , 5 , 5) is satisfied where time is "
   ↪ good time" then all{("xv" ; 'a is obligated to act for 'b
   ↪ after (15 , 5 , 5) where time is "agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
add (15 , 5 , 5) to "var 1" })
("xv" ; if nilaction every (15 , 5 , 5) starting at (5 , 5 , 5) and
   ↪  ending at (15 , 5 , 5) is satisfied where time is "good
   ↪ time" then all {noclauses})
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (5 , 5 , 5) and ending at (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then any {noclauses})
("xv" ; if do "string" by 'eko for 'ddn before (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then any {("xv" ; 'a is
   ↪ obligated to act for 'b after (15 , 5 , 5) where time is "
   ↪ agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
(subtract (15 , 5 , 5) from "var 2")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (5 , 5 , 5) and ending at (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then
any {noclauses})})
("xv" ; if nilaction at (15 , 5 , 5) is not satisfied where time is
   ↪  "good time" then all {("xv" ; 'a is obligated to act for 'b
   ↪  after (15 , 5 , 5) where time is "agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
```

```
   ↪  "agreement time")})
("xv" ; if nilaction every (15 , 5 , 5) starting at (5 , 5 , 5) and
   ↪  ending at (15 , 5 , 5) is not satisfied where time is "good
   ↪  time" then all {noclauses})
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (5 , 5 , 5) and ending at (15 , 5 , 5) is not
   ↪ satisfied where time is "good time" then any {noclauses})
("xv" ; if do "string" by 'eko for 'ddn before (15 , 5 , 5) is not
   ↪ satisfied where time is "good time" then any {("xv" ; 'a is
   ↪ obligated to act for 'b after (15 , 5 , 5) where time is "
   ↪ agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (5 , 5 , 5) and ending at (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then
any {noclauses})})}}) .
red get-contract-ancestors-names(Contract{
Preamble{
name: "My Contract"
start time: (15 , 5 , 5)
end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: "the address" , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address: "the
   ↪ address" , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address: "the
   ↪ address" , assets: [noassets]}]
extends: "abs" "ajs" "nada"}
Body{
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; if nilaction at (15 , 5 , 5) is satisfied where time is "
   ↪ good time" then all{("xv" ; 'a is obligated to act for 'b
   ↪ after (15 , 5 , 5) where time is "agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
```

```
add (15 , 5 , 5) to "var 1" })
("xv" ; if nilaction every (15 , 5 , 5) starting at (5 , 5 , 5) and
   ↪  ending at (15 , 5 , 5) is satisfied where time is "good
   ↪ time" then all {noclauses})
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (5 , 5 , 5) and ending at (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then any {noclauses})
("xv" ; if do "string" by 'eko for 'ddn before (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then any {("xv" ; 'a is
   ↪ obligated to act for 'b after (15 , 5 , 5) where time is "
   ↪ agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
(subtract (15 , 5 , 5) from "var 2")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (5 , 5 , 5) and ending at (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then
any {noclauses})})
("xv" ; if nilaction at (15 , 5 , 5) is not satisfied where time is
   ↪  "good time" then all {("xv" ; 'a is obligated to act for 'b
   ↪  after (15 , 5 , 5) where time is "agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")})
("xv" ; if nilaction every (15 , 5 , 5) starting at (5 , 5 , 5) and
   ↪  ending at (15 , 5 , 5) is not satisfied where time is "good
   ↪  time" then all {noclauses})
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (15 , 5 , 5) and ending at (15 , 5 , 5) is not
   ↪ satisfied where time is "good time" then any {noclauses})
("xv" ; if do "string" by 'eko for 'ddn before (12 , 5 , 5) is not
   ↪ satisfied where time is "good time" then any {
("xv" ; 'a is obligated to act for 'b after (15 , 5 , 5) where time
   ↪  is "agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
```

```
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (5 , 5 , 5) and ending at (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then
any {noclauses})})}}) .
quit .
```

B.9 Parties Module

```
load ../../specification/SLANC.maude .
red get-time-constraint("xv" ; 'a is obligated to act for 'b at (15
   ↪  , 5 , 5) where time is "agreement time") .
red get-variable-name("xv" ; 'a is obligated to act for 'b at (15 ,
   ↪  5 , 5) where time is "agreement time") .
quit .
```

B.10 Actions Module

```
load ../../specification/SLANC.maude .
parse nilaction @ (5 , 5 , 5) .
parse do "xyz" by 'elmo for 'ww @ 5 , 5 , 5 .
parse "xyz" by 'elmo for 'ww @ (5 , 5 , 5) .
parse transfer 3 SAR by 'abc for 'ee @ (5 , 5 , 5) .
parse (transfer 5 USD by 'nn for 'bb) @ (5 , 5 , 5) in fulfillment
   ↪ of "XVC" .
parse (transfer 5 USD by 'nn for 'bb) @ (5 , 5 , 5) in fulfillment
   ↪ of "DB" .
parse (transfer 5 USD by 'nn for 'bb) @ (5 , 5 , 5) in fulfillment
   ↪ of "NBC" .
quit .
```

B.11 Assets Module

```
load ../../specification/SLANC.maude .
parse { id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ "the address" , assets: [
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }
```

```
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }] } .
parse { id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ "the address" , assets: [noassets] }
{ id: 'abs , name: "abssss" , description: "deco" , address: "the
   ↪ address" , assets: [
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }] } .
parse { id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ "the address" , assets: [
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }] }
{ id: 'abs , name: "abssss" , description: "deco" , address: "the
   ↪ address" , assets: [noassets] }
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [{ name: "Saudi Ryials", code: SAR ,
   ↪ supply: 500000, owner: 'ElKoko }
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }
{ name: "Saudi Ryials", code: SAR , supply: 500000, owner: 'ElKoko
   ↪ }] } .
quit .
```

B.12 Preamble Module

```
load ../../specification/SLANC.maude .
parse Preamble .
```

```
parse Preamble{
name: "My Contract"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}]
extends: none} .
parse Preamble{
name: "My Contract"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}]
extends: none} .
parse Preamble{
name: "My Contract"
start time: (15 , 5 , 5) end time: (15 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress , assets: [noassets]}]
extends: none} .
quit .
```

B.13 Rules Module

```
load ../../specification/SLANC.maude .
parse "xv" ; if nilaction at (15 , 5 , 5) is satisfied where time
   ↪ is "good time" then all {("xv" ; 'a is permitted to act for
   ↪ 'b after (15 , 5 , 5) where time is "agreement time")
("xv" ; 'a is permitted to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
```

```
   ↪  "agreement time")
add (15 , 5 , 5) to "var 1" } .
parse "xv" ; if nilaction every (15 , 5 , 5) starting at (15 , 5 ,
   ↪ 5) and ending at (15 , 5 , 5) is satisfied where time is "
   ↪ good time" then all {noclauses} .
parse "xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (15 , 5 , 5) and ending at (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then any {noclauses} .
parse "xv" ; if do "string" by 'eko for 'ddn before (15 , 5 , 5) is
   ↪  satisfied where time is "good time" then any {("xv" ; 'a is
   ↪  obligated to act for 'b after (15 , 5 , 5) where time is "
   ↪ agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is prohibited from act for 'b at (15 , 5 , 5) where time
   ↪  is "agreement time")
(subtract (15 , 5 , 5) from "var 2")
("xv" ; 'a is prohibited from act for 'b at (15 , 5 , 5) where time
   ↪  is "agreement time")
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (15 , 5 , 5) and ending at (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then
any {noclauses})} .
parse "xv" ; if nilaction at (15 , 5 , 5) is not satisfied where
   ↪ time is "good time" then all {("xv" ; 'a is obligated to act
   ↪  for 'b after (15 , 5 , 5) where time is "agreement time")
("xv" ; 'a is obligated to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")} .
parse "xv" ; if nilaction every (15 , 5 , 5) starting at (15 , 5 ,
   ↪ 5) and ending at (15 , 5 , 5) is not satisfied where time is
   ↪  "good time" then all {noclauses} .
parse "xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (15 , 5 , 5) and ending at (15 , 5 , 5) is not
   ↪ satisfied where time is "good time" then any {noclauses} .
parse "xv" ; if do "string" by 'eko for 'ddn before (15 , 5 , 5) is
   ↪  not satisfied where time is "good time" then any {("xv" ; '
   ↪ a is obligated to act for 'b after (15 , 5 , 5) where time
   ↪ is "agreement time")
("xv" ; 'a is permitted to act for 'b between (15 , 5 , 5) and "
   ↪ play time" where time is "agreement time")
("xv" ; 'a is permitted to act for 'b at (15 , 5 , 5) where time is
```

```
   ↪  "agreement time")
("xv" ; 'a is permitted to act for 'b at (15 , 5 , 5) where time is
   ↪  "agreement time")
("xv" ; if transfer 11 USD by 'nn for 'bb every (15 , 5 , 5)
   ↪ starting at (15 , 5 , 5) and ending at (15 , 5 , 5) is
   ↪ satisfied where time is "good time" then
any {noclauses})} .
quit .
```

B.14 State Module

```
load ../../specification/SLANC.maude .
parse noglobalstate .
parse nilstatement .
parse noclausestate .
parse (agreement "xvc" is breached at (5 , 5 , 5)) (agreement "xvc"
   ↪  is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5)) .
parse "ccv" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (
   ↪ agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))} .
parse ("ccv" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (
   ↪ agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})
("ccv" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (agreement "
   ↪ xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})
("ccv" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (agreement "
   ↪ xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))}) .
red find-state-of(("del" ; 'a is obligated to act for 'b at (15 , 5
   ↪  , 5) where time is "agreement time"),("ccv" ; {(agreement "
   ↪ xvc" is breached at (5 , 5 , 5)) (agreement "xvc" is
   ↪ breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})
("xvc" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (agreement "
   ↪ xvc" is breached at (5 , 5 , 5))
```

```
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})
("ccv" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (agreement "
   ↪ xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})) .
red find-state-of(("xvc" ; 'a is obligated to act for 'b at (15 , 5
   ↪  , 5) where time is "agreement time"),("ccv" ; {(agreement "
   ↪ xvc" is breached at (5 , 5 , 5)) (agreement "xvc" is
   ↪ breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})
("xvc" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (agreement "
   ↪ xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})
("ccv" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (agreement "
   ↪ xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})) .
red update-state-of(("del" ; 'a is obligated to act for 'b at (15 ,
   ↪  5 , 5) where time is "agreement time"),nostatements,("ccv"
   ↪ ; {(agreement "xvc" is breached at (5 , 5 , 5)) (agreement "
   ↪ xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})
("xvc" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (agreement "
   ↪ xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})
("ccv" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (agreement "
   ↪ xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})) .
red update-state-of(("ccc" ; 'a is obligated to act for 'b at (15 ,
   ↪  5 , 5) where time is "agreement time")
,(agreement "ccc" is breached at (5 , 5 , 5)) (agreement "ccc" is
   ↪ breached at (5 , 5 , 5))
(agreement "ccc" is breached at (5 , 5 , 5))
,("ccv" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (agreement
   ↪ "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})
("ccc" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (agreement "
   ↪ xvc" is breached at (5 , 5 , 5))
```

```
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})
("ccv" ; {(agreement "xvc" is breached at (5 , 5 , 5)) (agreement "
   ↪ xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))
(agreement "xvc" is breached at (5 , 5 , 5))})) .
quit .
load ../../specification/SLANC.maude .
red "a" in "ab" "c" "d" "e" "x" .
quit .
```

B.15 Violation Detection Semantics Module

```
load ../../specification/SLANC.maude .
red (30 , 1 , 2018) ++ (1 , 0 , 0) .
red (31 , 1 , 2018) ++ (1 , 0 , 0) .
red (31 , 1 , 2018) ++ (1 , 0 , 0) .
red (1 , 2 , 2018) ++ (1 , 0 , 0) .
red (5 , 5 , 5) ++ ( 11 , 11 , 11) .
red lte((31 , 12 , 2018), (1 , 1 , 2019)) .
red lte((1 , 1 , 2019),(31 , 12 , 2018) ) .
red count-increments(every (1 , 0 , 0) starting at (1 , 1 , 2018 )
   ↪ and ending at (31 , 12 , 2018 ),
("Start Time" is (1 , 1 , 2018 ))("End Time" is (31 , 12 , 2018 )))
   ↪  .
red count-increments(every (1 , 0 , 0) starting at (1 , 1 , 2018 )
   ↪ and ending at (31 , 12 , 2018 ),
("Start Time" is (1 , 1 , 2018 ))("End Time" is (31 , 12 , 2018 )))
   ↪  .
red (1 , 0 , 0) ++ (29 , 2 , 2018) .
red (1 , 12 , 2018) ++ ( 1 , 0 , 0) .
red lt(undefined-time,(1 , 3 , 2018)) .
red undefined-time ++ (1 , 3 , 2018) .
quit .
```

B.16 Variables Module

```
load ../../specification/SLANC.maude .
parse "x" is (15 , 5 , 5) .
parse "x" # 4 is (15 , 5 , 5) .
parse "x" # 5 is (15 , 5 , 5) .
parse "x" # 6 is (15 , 5 , 5) .
```

```
parse "first party" is 'wlkoko .
parse where time is "first time" .
parse ( "bbb" ; add (15 , 5 , 5) to "a") ( "ccc" ; subtract (15 , 5
   ↪  , 5) from "b") .
red update-with-variable-update-statement(( "Kkk" ; add (15 , 5 ,
   ↪ 5) to "ssa" ) ,
"ddd" by 'v for 'n @ (1 , 2 , 3), (noassets | ("ssa" is (1 , 1 , 1)
   ↪ ) | noglobalstate | nopairs | notransactions) ) .
red evaluate-time-arithmatic(("xvc" + (15 , 5 , 5)),novariables) .
red evaluate-time-arithmatic(("xvc" + (15 , 5 , 5)),"xvc" is (15 ,
   ↪ 5 , 5)) .
red evaluate-time-arithmatic(("xvc" + (1 , , 5)),"xvc" is (15 , 5 ,
   ↪  5)) .
red evaluate-time-arithmatic((("xvc" # 5) + (15 , 5 , 5)),"xvc" is
   ↪ (15 , 5 , 5) ) .
red evaluate-time-arithmatic((("xvc" # 5) + (15 , 5 , 5)),("xvc" is
   ↪  (15 , 5 , 5)) ("xvc" # 5 is (15 , 5 , 5)) ) .
red get-variable-group("XBC",novariables) .
red get-variable-group("XBC", ("xvc" is (15 , 5 , 5))
(("xvc" # 5 is (15 , 5 , 5))
("XBC" group is {("xvc" is (15 , 5 , 5))
("xvc" # 5 is (15 , 5 , 5))
})) ) .
red count-variables(("xvc" is (15 , 5 , 5))
("xvc" # 5 is (15 , 5 , 5))
("XBC" group is {("xvc" is (15 , 5 , 5))
("xvc" # 5 is (15 , 5 , 5))
})) .
red get-variable-list(("XBC" group is {("xvc" is (15 , 5 , 5))
("xvc" # 5 is (15 , 5 , 5))
})) .
red heads-time(("xvc" is (15 , 5 , 5))
(("xvc" # 5 is (15 , 5 , 5))
("XBC" group is {("xvc" is (15 , 5 , 5))
("xvc" # 5 is (15 , 5 , 5))
}))) .
red update-variable-group("XBC","A" # 5 is (1 , 1 , 1), ("xvc" is
   ↪ (15 , 5 , 5))
(("xvc" # 5 is (15 , 5 , 5))
("XBC" group is {("xvc" is (15 , 5 , 5))
("xvc" # 5 is (15 , 5 , 5))
})) ) .
red get-variable-name("A" + (15 , 5 , 5)) .
red get-single-variable-time-value("xvc",(("xvcxx" # 5 is (15 , 5 ,
   ↪  5))
```

```
("XBC" group is {("xvc" is (15 , 5 , 5))
("xvc" # 5 is (15 , 5 , 5))
}) ("xvc" is (145 , 5 , 5)))) .
red evaluate-time-arithmatic("lessee termination notice" + (0 , 1 ,
   ↪  0) , "lessee termination notice" is (30 , 11 ,2018) ) .
quit
```

B.17 Violation Detection Semantics Module

```
load ../../specification/SLANC.maude .
red update-clause-state-given-event("xv" ; 'a is obligated to do "
   ↪ abc" for 'b at (19 , 5 , 5) where time is "agreement time"
   ↪ ,(do "abc" by 'a for 'b) @ (15 , 5 , 5),(noassets |
   ↪ novariables | noglobalstate | nopairs | notransactions)) .
red update-clause-state-given-event("xv" ; 'a is prohibited from do
   ↪  "abc" for 'b at (15 , 5 , 5) where time is "agreement time"
   ↪  ,(do "abc" by 'a for 'b) @ (15 , 5 , 5) ,(noassets |
   ↪ novariables | noglobalstate | nopairs | notransactions)) .
red update-body-state-given-environment(Body { ("xv" ; 'a is
   ↪ prohibited from do "abc" for 'b at (15 , 5 , 5) where time
   ↪ is "agreement time") },((do "abc" by 'a for 'b) @ (15 , 5 ,
   ↪ 5)) ((do "abc" by 'a for 'b) @ (15 , 5 , 6)),(noassets |
   ↪ novariables | noglobalstate | nopairs | notransactions)) .
red update-body-state-given-event(Body {"xv" ; 'a is prohibited
   ↪ from do "abc" for 'b at (15 , 5 , 5) where time is "
   ↪ agreement time" },(do "abc" by 'a for 'b) @ (15 , 5 , 5) ,(
   ↪ noassets | novariables | noglobalstate | nopairs |
   ↪ notransactions)) .
red update-clause-state-given-event("xv" ; 'a is permitted to do "
   ↪ abc" for 'b at (15 , 5 , 5) where time is "agreement time"
   ↪ ,(do "abc" by 'a for 'b) @ (15 , 5 , 5),(noassets |
   ↪ novariables | noglobalstate | nopairs | notransactions)) .
red update-clause-state-given-event("ag" ; 'a is obligated to do "
   ↪ abc" for 'b after (15 , 5 , 5) where time is "agreement time
   ↪ " ,(do "abc" by 'a for 'b) @ (18 , 5 , 5),(noassets |
   ↪ novariables | noglobalstate | nopairs | notransactions)) .
red update-clause-state-given-event("xv" ; if do "abc" by 'a for 'b
   ↪  before (15 , 5 , 5) is satisfied where time is "good time"
   ↪ then any {("xv" ; 'a is obligated to act for 'b after (15 ,
   ↪ 5 , 5) where time is "agreement time") } ,(do "abc" by 'a
   ↪ for 'b) @ (15 , 5 , 5),(noassets | novariables |
   ↪ noglobalstate | nopairs | notransactions)) .
red update-clause-state-given-event("rl" ; if do "abc" by 'a for 'b
```

```
   ↪  at (15 , 5 , 5) is satisfied where time is "good time" then
   ↪  any {("ag" ; 'a is obligated to do "abc" for 'b before (16
   ↪ , 5 , 5) where time is "agreement time1") ("ag" ; 'a is
   ↪ obligated to do "abc" for 'b at (16 , 5 , 5) where time is "
   ↪ agreement time2") } ,(do "abc" by 'a for 'b) @ (15 , 5 , 5)
   ↪ ,(noassets | novariables | noglobalstate | nopairs |
   ↪ notransactions)) .
red update-clause-state-given-event("rl" ; if do "abc" by 'a for 'b
   ↪  at (15 , 5 , 5) is satisfied where time is "good time" then
   ↪  any {("ag" ; 'a is obligated to do "abc" for 'b before (16
   ↪ , 5 , 5) where time is "agreement time1") ("ag" ; 'a is
   ↪ obligated to do "abc" for 'b at (16 , 5 , 5) where time is "
   ↪ agreement time2") } ,(do "abc" by 'a for 'b) @ (15 , 5 , 5)
   ↪ ,(noassets | novariables | noglobalstate | nopairs |
   ↪ notransactions)) .
red update-clause-state-given-event("rl" ; if do "abc" by 'a for 'b
   ↪  at (15 , 5 , 5) is satisfied where time is "good time" then
   ↪  all {("ag" ; 'a is obligated to do "abc" for 'b before (16
   ↪ , 5 , 5) where time is "agreement time1") } ,(do "abc" by 'a
   ↪  for 'b) @ (15 , 5 , 5),(noassets | novariables |
   ↪ noglobalstate | nopairs | notransactions)) .
red update-clause-state-given-event("rl" ; if do "abc" by 'a for 'b
   ↪  at (15 , 5 , 5) is satisfied where time is "good time" then
   ↪  all {("ag" ; 'a is obligated to do "abc" for 'b before (16
   ↪ , 5 , 5) where time is "agreement time1") ("ag2" ; 'a is
   ↪ obligated to do "abc" for 'b at (19 , 5 , 5) where time is "
   ↪ agreement time2") } ,(do "abc" by 'a for 'b) @ (15 , 5 , 5)
   ↪ ,(noassets | novariables | noglobalstate | nopairs |
   ↪ notransactions)) .
red update-body-state-given-environment(Body { ("rl" ; if do "abc"
   ↪ by 'a for 'b at (15 , 5 , 5) is satisfied where time is "
   ↪ good time" then all {("ag" ; 'a is obligated to do "abc" for
   ↪  'b before (16 , 5 , 5) where time is "agreement time1") ("
   ↪ ag2" ; 'a is obligated to do "abc" for 'b at (19 , 5 , 5)
   ↪ where time is "agreement time2") }) } ,((do "abc" by 'a for
   ↪ 'b) @ (15 , 5 , 5)) (do "abc" by 'a for 'b) @ (19 , 5 , 5),(
   ↪ noassets | novariables | noglobalstate | nopairs |
   ↪ notransactions)) .
red update-clause-state-given-event("cnt" ; any where time is "koko
   ↪ " {("ag" ; 'a is obligated to do "abc" for 'b before (16 , 5
   ↪  , 5) where time is "agreement time") } ,(do "abc" by 'a for
   ↪  'b) @ (15 , 5 , 5),(noassets | novariables | noglobalstate
   ↪ | nopairs | notransactions)) .
red update-body-state-given-environment(Body { ("ag" ; 'a is
   ↪ obligated to do "abc" for 'b every (1 , 0 , 0) starting at
```

```
   ↪ (16 , 5 , 5) and ending at (18 , 5 , 5) where time is "
   ↪ agreement time1") } ,
((do "abc" by 'a for 'b) @ (15 , 5 , 5))
((do "abc" by 'a for 'b) @ (16 , 5 , 5))
((do "abc" by 'a for 'b) @ (17 , 5 , 5))
((do "abc" by 'a for 'b) @ (18 , 5 , 5)),(noassets | novariables |
   ↪ noglobalstate | nopairs | notransactions)) .
red update-body-state-given-environment(Body { ("ag" ; 'a is
   ↪ obligated to do "abc" for 'b every (1 , 0 , 0) starting at
   ↪ (16 , 5 , 5) and ending at (18 , 5 , 5) where time is "
   ↪ agreement time1") ("ag2" ; 'a is obligated to do "abc" for '
   ↪ b at each "agreement time1" where time is "agreement time2")
   ↪  } ,
((do "abc" by 'a for 'b) @ (15 , 5 , 5))
((do "abc" by 'a for 'b) @ (16 , 5 , 5))
((do "abc" by 'a for 'b) @ (17 , 5 , 5))
((do "abc" by 'a for 'b) @ (18 , 5 , 5))
((do "abddc" by 'a for 'b) @ (19 , 5 , 5)),(noassets | novariables
   ↪ | noglobalstate | nopairs | notransactions)) .
red count-increments(every (1 , 0 , 0) starting at (16 , 5 , 5) and
   ↪  ending at (18 , 5 , 5),novariables) .
red update-body-state-given-environment(Body { ("ag" ; 'a is
   ↪ prohibited from do "abc" for 'b every (1 , 0 , 0) starting
   ↪ at (16 , 5 , 5) and ending at (18 , 5 , 5) where time is "
   ↪ agreement time1") ("ag2" ; 'a is obligated to do "abc" for '
   ↪ b at each "agreement time1" where time is "agreement time2")
   ↪  } ,
((do "abc" by 'a for 'b) @ (15 , 5 , 5))
((do "abc" by 'a for 'b) @ (16 , 5 , 5))
((do "abc" by 'a for 'b) @ (17 , 5 , 5))
((do "abc" by 'a for 'b) @ (18 , 5 , 5))
((do "abddc" by 'a for 'b) @ (19 , 5 , 5)),(noassets | novariables
   ↪ | noglobalstate | nopairs | notransactions)) .
red update-body-state-given-environment(Body { ("ag" ; 'a is
   ↪ prohibited from do "abc" for 'b every (1 , 0 , 0) starting
   ↪ at (16 , 5 , 5) and ending at (18 , 5 , 5) where time is "
   ↪ agreement time1") ("ag2" ; 'a is obligated to do "abcc" for
   ↪ 'b at each "agreement time1" where time is "agreement time2
   ↪ ") } ,
((do "abcc" by 'a for 'b) @ (15 , 5 , 5))
((do "abc" by 'a for 'b) @ (15 , 5 , 5))
((do "abc" by 'a for 'b) @ (16 , 5 , 5))
((do "abcc" by 'a for 'b) @ (16 , 5 , 5))
((do "abc" by 'a for 'b) @ (17 , 5 , 5))
((do "abcc" by 'a for 'b) @ (17 , 5 , 5))
```

```
((do "abddc" by 'a for 'b) @ (19 , 5 , 5)),(noassets | novariables
   ↪ | noglobalstate | nopairs | notransactions)) .
red update-body-state-given-environment(Body { ("rl" ; if do "abc"
   ↪ by 'a for 'b every (1 , 0 , 0) starting at (16 , 5 , 5) and
   ↪ ending at (18 , 5 , 5) is satisfied where time is "rl1 time"
   ↪  then all { ("ag12" ; 'a is obligated to do "abc" for 'b at
   ↪ each "rl1 time" where time is "agreement time12") })} ,
((do "abc" by 'a for 'b) @ (15 , 5 , 5))
((do "abc" by 'a for 'b) @ (16 , 5 , 5))
((do "abc" by 'a for 'b) @ (17 , 5 , 5))
((do "abc" by 'a for 'b) @ (18 , 5 , 5)),(noassets | novariables |
   ↪ noglobalstate | nopairs | notransactions)) .
red update-body-state-given-environment(Body { ("rl" ; if do "abc"
   ↪ by 'a for 'b every (1 , 0 , 0) starting at (16 , 5 , 5) and
   ↪ ending at (18 , 5 , 5) is satisfied where time is "rl1 time"
   ↪  then all { ("ag12" ; 'a is obligated to do "abc" for 'b at
   ↪ each "rl1 time" where time is "agreement time12") })} ,
((do "abc" by 'a for 'b) @ (15 , 5 , 5))
((do "abc" by 'a for 'b) @ (16 , 5 , 5))
((do "abc" by 'a for 'b) @ (17 , 5 , 5))
((do "abc" by 'a for 'b) @ (18 , 5 , 5))
((do "abc" by 'a for 'b) @ (19 , 5 , 5))
((do "abc" by 'a for 'b) @ (19 , 5 , 5))
((do "abc" by 'a for 'b) @ (19 , 5 , 5))
((do "abc" by 'a for 'b) @ (19 , 5 , 5))
((do "abc" by 'a for 'b) @ (19 , 5 , 5))
((do "abc" by 'a for 'b) @ (19 , 5 , 5))
((do "abc" by 'a for 'b) @ (19 , 5 , 5)),(noassets | novariables |
   ↪ noglobalstate | nopairs | notransactions)) .
red update-body-state-given-environment(Body { ("cc" ; all where
   ↪ time is "good time"{ ("ag12" ; 'a is obligated to do "abc"
   ↪ for 'b at (15 , 5 , 5) where time is "agreement time12") ("
   ↪ ag13" ; 'a is obligated to do "abcd" for 'b at (16 , 5 , 5)
   ↪ where time is "agreement time13") } )} ,
((do "abc" by 'a for 'b) @ (15 , 5 , 5))
((do "abcd" by 'a for 'b) @ (16 , 5 , 5)),(noassets | novariables |
   ↪  noglobalstate | nopairs | notransactions)) .
red update-body-state-given-environment(Body { ("cc" ; any where
   ↪ time is "good time"{ ("ag12" ; 'a is obligated to do "abc"
   ↪ for 'b at (15 , 5 , 5) where time is "agreement time12") ("
   ↪ ag13" ; 'a is obligated to do "abcd" for 'b at (16 , 5 , 5)
   ↪ where time is "agreement time13") } )} ,
((do "abc" by 'a for 'b) @ (15 , 5 , 5))
((do "abcd" by 'a for 'b) @ (16 , 5 , 5)),(noassets | novariables |
```

```
   ↪  noglobalstate | nopairs | notransactions)) .
red update-body-state-given-environment(Body { ("cc" ; sequence
   ↪ where time is "good time"{ ("ag12" ; 'a is obligated to do "
   ↪ abc" for 'b at (15 , 5 , 5) where time is "agreement time12
   ↪ ") ("ag13" ; 'a is obligated to do "abcd" for 'b at (16 , 5
   ↪ , 5) where time is "agreement time13") } )} ,
((do "abc" by 'a for 'b) @ (15 , 5 , 5))
((do "abcd" by 'a for 'b) @ (16 , 5 , 5)),(noassets | novariables |
   ↪  noglobalstate | nopairs | notransactions)) .
red update-body-state-given-environment(Body { ("cc" ; parallel
   ↪ where time is "good time"{ ("ag12" ; 'a is obligated to do "
   ↪ abc" for 'b at (15 , 5 , 5) where time is "agreement time12
   ↪ ") ("ag13" ; 'a is obligated to do "abcd" for 'b at (16 , 5
   ↪ , 5) where time is "agreement time13") } )} ,
((do "abc" by 'a for 'b) @ (15 , 5 , 5))
((do "abcd" by 'a for 'b) @ (16 , 5 , 5)),(noassets | novariables |
   ↪  noglobalstate | nopairs | notransactions)) .
red update-clause-state-given-event("ag" ; 'a is obligated to do "
   ↪ abc" for 'b at (15 , 5 , 5) where time is "agreement time"
   ↪ ,(do "abc" by 'a for 'b) @ (19 , 5 , 5),(noassets |
   ↪ novariables | noglobalstate | nopairs | notransactions)) .
red detect-violations(Contract{
Preamble{
name: "My Contract1"
start time: (15 , 5 , 5)
end time: (18 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress}]
extends: "My Contract"}
Body{
("xv1" ; 'a is obligated to act for 'b at (15 , 5 , 5) where time
   ↪ is "agreement time1")}},("abcc" by 'a for 'b @ (18 , 5, 5)),
Contract{
Preamble{
name: "My Contract"
start time: (15 , 5 , 5)
end time: (19 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress }
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress }]
extends: none}
Body{
```

```
("xv" ; 'a is obligated to act for 'b at (16 , 5 , 5) where time is
   ↪  "agreement time2")}}) .
red detect-violations(Contract{
Preamble{
name: "My Contract1"
start time: (15 , 5 , 5)
end time: (18 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress}
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress}]
extends: "My Contract"}
Body{
(override ("xva" ; 'a is obligated to act for 'b at (15 , 5 , 5)
   ↪ where time is "agreement time1"))}
},("abcc" by 'a for 'b @ (18 , 5, 5)), (Contract{
Preamble{
name: "My Contract"
start time: (15 , 5 , 5)
end time: (19 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress }
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress }]
extends: "My Contracta"}
Body{
("xv" ; 'a is obligated to act for 'b at (16 , 5 , 5) where time is
   ↪  "agreement time2")}})
Contract{
Preamble{
name: "My Contracta"
start time: (15 , 5 , 5)
end time: (19 , 5 , 5)
parties: [{ id: 'abs , name: "abssss" , description: "deco" ,
   ↪ address: noaddress }
{ id: 'abs , name: "abssss" , description: "deco" , address:
   ↪ noaddress }]
extends: none}
Body{
("xva" ; 'a is obligated to act for 'b at (16 , 5 , 5) where time
   ↪ is "agreement time2a")}}) .
red update-clause-state-given-event(
("1. Items Left in Vehicle" ;
'Lessor is not obligated to
"replace damaged items left in lessee's car"
```

```
for 'Lessee at all times where time is "item damage time")
, ( ("replace damaged items left in lessee's car" by 'Lessor for '
   ↪ Lessee) @ (31 , 1 , 2018) ) ,
( noassets | ("Start Time" is (1 , 1 , 2018)) ("End Time" is (1 , 2
   ↪  , 2018)) | noglobalstate | nopairs | notransactions )) .
red update-clause-state-given-event(
("1. Items Left in Vehicle" ;
'Lessor is obligated to
"replace damaged items left in lessee's car"
for 'Lessee every (1 , 0 , 0) starting at "Start Time" and ending
   ↪ at "End Time" where time is "item damage time")
, ( ("replace damaged items left in lessee's car" by 'Lessor for '
   ↪ Lessee) @ (2 , 1 , 2018) ) ,
( noassets | ("Start Time" is (1 , 1 , 2018)) ("End Time" is (2 , 1
   ↪  , 2018)) | noglobalstate | nopairs | notransactions )) .
red update-clause-state-given-event(
("1. Items Left in Vehicle" ;
'Lessor is obligated to
"replace damaged items left in lessee's car"
for 'Lessee at any time repeated where time is "item damage time")
, ( ("replace damaged items left in lessee's car" by 'Lessor for '
   ↪ Lessee) @ (3 , 1 , 2018) ) ,
( noassets | ("Start Time" is (1 , 1 , 2018)) ("End Time" is (2 , 1
   ↪  , 2018)) | noglobalstate | nopairs | notransactions )) .
red update-body-state-given-environment(Body { ("agr" ; 'a is
   ↪ obligated to "abcs" for 'b at any time repeated where time
   ↪ is "agr time")} ,
(("abc" by 'a for 'b) @ (15 , 5 , 5))
(("abc" by 'a for 'b) @ (16 , 5 , 5))
(("abc" by 'a for 'b) @ (17 , 5 , 5))
(("abc" by 'a for 'b) @ (18 , 5 , 5))
(("abc" by 'a for 'b) @ (19 , 5 , 5)),
(noassets | ("Start Time" is (15 , 5 , 5)) ("End Time" is (18 , 5 ,
   ↪  5)) | noglobalstate | nopairs | notransactions)) .
red update-clause-state-given-event(
("7. Damages and Loss of Equipment." ; any where time is "damage
   ↪ and fix time" {
("7.1 Damage to parking facilities" ; 'Lessee is prohibited from "
   ↪ damage parking facilities" for 'Lessor at any time repeated
   ↪ where time is "parking damage time")
("7.2 Fixing damages to parking facilities" ; 'Lessee is obligated
   ↪ to "fix damage to parking facilities" for 'Lessor at each "
   ↪ parking damage time" where time is "parking fix time")})
, ( ("replace damaged items left in lessee's car" by 'Lessor for '
   ↪ Lessee) @ (3 , 1 , 2018) ) ,
```

```
( noassets | ("Start Time" is (1 , 1 , 2018)) ("End Time" is (8 , 1
   ↪  , 2018)) | noglobalstate | nopairs | notransactions )) .
red container-satisfied(all,
("lessee termination clause" ; 'Lessee is permitted to "terminate
   ↪ agreement" for 'Lessor
at ("lessee termination notice" + (0 , 1 , 0)) where time is "
   ↪ Lessee termination time"),
("lessee termination clause" fulfilled by (1 , 1 , 1) with time
   ↪ recorded in "aaa")
) .
red is-variable-count-match("less",at any time,"less" is (1 , 2 , 3
   ↪  )) .
quit .
```

REFERENCES

[1] W. M. Farmer and Q. Hu, "A formal language for writing contracts," in *2016 IEEE 17th International Conference on Information Reuse and Integration (IRI)*, July 2016, pp. 134–141.

[2] S. L. P. Jones, "Composing contracts: An adventure in financial engineering," in *FME*, vol. 2021, 2001, p. 435.

[3] M. Palmirani, G. Governatori, A. Rotolo, S. Tabet, H. Boley, and A. Paschke, "Legalruleml: Xml-based rules and norms," in *Rule-Based Modeling and Computing on the Semantic Web*. Springer, 2011, pp. 298–312.

[4] L. E. Allen, "Symbolic logic: A razor-edged tool for drafting and interpreting legal documents," in *Faculty Scholarship Series*. Yale School of Law, 1957, paper 4519.

[5] B. A. Blum, *Contracts: examples & explanations*. Aspen Publishers Online, 2007.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude - A High-Performance Logical Framework*, ser.

LNCS. Secaucus, NJ, USA: Springer-Verlag, 2007, vol. 4350.

[7] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer, "Specification and proof in membership equational logic," *Theoretical Computer Science*, vol. 236, pp. 35–132, 2000.

[8] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theor. Comput. Sci.*, vol. 96, no. 1, pp. 73–155, 1992.

[9] B.-Y. Wang, J. Meseguer, and C. A. Gunter, "Specification and formal analysis of a plan algorithm in maude." in *ICDCS Workshop on Distributed System Validation and Verification*, 2000, pp. E49–E56.

[10] M. Hicks, J. T. Moore, D. S. Alexander, C. A. Gunter, and S. M. Nettles, "Planet: An active internetwork," in *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 1999, pp. 1124–1133.

[11] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles, "Plan: A packet language for active networks," in *ACM SIGPLAN Notices*, vol. 34, no. 1. ACM, 1998, pp. 86–93.

[12] J. E. Rivera, F. Durán, and A. Vallecillo, "Formal specification and analysis of domain specific models using maude," *Simulation*, vol. 85, no. 11-12, pp. 778–792, 2009.

[13] J. Troya and A. Vallecillo, "Towards a rewriting logic semantics for atl," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2010, pp. 230–244.

[14] A. Elgammal, O. Turetken, W.-J. van den Heuvel, and M. Papazoglou, "On the formal specification of regulatory compliance: a comparative analysis," in *International Conference on Service-Oriented Computing*. Springer, Berlin, Heidelberg, 2010, pp. 27–38.

[15] T. Hvitved, "A survey of formal languages for contracts," *Formal Language and Analysis of Contract-Oriented Software*, 2010.

[16] G. Governatori and Z. Milosevic, "A formal analysis of a business contract language," *International Journal of Cooperative Information Systems*, vol. 15, no. 04, pp. 659–685, 2006.

[17] G. Pace, C. Prisacariu, and G. Schneider, "Model checking contracts-a case study," in *ATVA*, vol. 7, 2007, pp. 82–97.

[18] C. Prisacariu and G. Schneider, "A formal language for electronic contracts," in *FMOODS*, vol. 7, 2007, pp. 174–189.

[19] G. J. Pace and M. Rosner, "A controlled language for the specification of contracts." *CNL*, vol. 9, pp. 226–245, 2009.

[20] C. Prisacariu and G. Schneider, "\mathcal {CL}: An action-based logic for reasoning about contracts," in *International Workshop on Logic, Language,*

Information, and Computation. Springer, Berlin, Heidelberg, 2009, pp. 335–349.

[21] T. Hvitved, F. Klaedtke, and E. Zălinescu, "A trace-based model for multi-party contracts," *The Journal of Logic and Algebraic Programming*, vol. 81, no. 2, pp. 72–98, 2012.

[22] K. Angelov, J. J. Camilleri, and G. Schneider, "A framework for conflict analysis of normative texts written in controlled natural language," *The Journal of Logic and Algebraic Programming*, vol. 82, no. 5-7, pp. 216–240, 2013.

[23] M. Cambronero, L. Llana, and G. Pace, "A Calculus Supporting Contract Reasoning and Monitoring," *IEEE Access*, vol. 5, no. c, pp. 1–10, 2017.

[24] S. Berthold, "Towards a formal language for privacy options." in *PrimeLife*, 2010, pp. 27–40.

[25] S. P. Jones and J.-M. Eber, "How to write a financial contract," 2003.

[26] D.-I. J.-M. Gaillourdet, "A software language approach to derivative contracts in finance," *YRS 2011*, p. 39, 2011.

[27] S. Peyton Jones, J.-M. Eber, and J. Seward, "Composing contracts: an adventure in financial engineering (functional pearl)," in *ACM SIGPLAN Notices*, vol. 35, no. 9. ACM, 2000, pp. 280–292.

[28] P. Bahr, J. Berthold, and M. Elsman, "Certified symbolic management of financial multi-party contracts," *ACM SIGPLAN Notices*, vol. 50, no. 9, pp. 315–327, 2015.

[29] S. Peyton Jones, J.-M. Eber, and J. Seward, "Composing contracts: An adventure in financial engineering (functional pearl)," *SIGPLAN Not.*, vol. 35, no. 9, pp. 280–292, Sep. 2000. [Online]. Available: http://doi.acm.org/10.1145/357766.351267

[30] J. A. E. E. F. Hengleina and J. G. S. C. Stefansena, "Compositional specification of commercial contracts," *Stefansen, Christian Oskar Erik*, p. 54, 2008.

[31] T. Hvitved, "A survey of formal languages for contracts," in *Fourth Workshop on Formal Languages and Analysis of Contract–Oriented Software (FLACOS'10)*, 2010, pp. 29–32.

VITAE

- Name: Turki Saud Alhazmi
- Nationality: Saudi
- Date of Birth: 10/11/1991
- Email: *hazmi.ts@gmail.com*
- Permanent Address: Medina 42391, Saudi Arabia
- Education Background:
  - Master of Science in Computer Science,

    King Fahd University of Petroleum and Minerals, May 2018.
  - Bachelors of Science in Computer Science,

    King Fahd University of Petroleum and Minerals, Feb 2015.